<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.8</td>
<td>Debugging memory leaks</td>
<td>148</td>
</tr>
<tr>
<td>5.9</td>
<td>Downloading and processing files and images</td>
<td>152</td>
</tr>
<tr>
<td>5.10</td>
<td>Deploying Spiders</td>
<td>158</td>
</tr>
<tr>
<td>5.11</td>
<td>AutoThrottle extension</td>
<td>159</td>
</tr>
<tr>
<td>5.12</td>
<td>Benchmarking</td>
<td>161</td>
</tr>
<tr>
<td>5.13</td>
<td>Jobs: pausing and resuming crawls</td>
<td>162</td>
</tr>
<tr>
<td>6</td>
<td>Extending Scrapy</td>
<td>165</td>
</tr>
<tr>
<td>6.1</td>
<td>Architecture overview</td>
<td>165</td>
</tr>
<tr>
<td>6.2</td>
<td>Downloader Middleware</td>
<td>168</td>
</tr>
<tr>
<td>6.3</td>
<td>Spider Middleware</td>
<td>181</td>
</tr>
<tr>
<td>6.4</td>
<td>Extensions</td>
<td>185</td>
</tr>
<tr>
<td>6.5</td>
<td>Core API</td>
<td>190</td>
</tr>
<tr>
<td>6.6</td>
<td>Signals</td>
<td>198</td>
</tr>
<tr>
<td>6.7</td>
<td>Item Exporters</td>
<td>202</td>
</tr>
<tr>
<td>7</td>
<td>All the rest</td>
<td>209</td>
</tr>
<tr>
<td>7.1</td>
<td>Release notes</td>
<td>209</td>
</tr>
<tr>
<td>7.2</td>
<td>Contributing to Scrapy</td>
<td>247</td>
</tr>
<tr>
<td>7.3</td>
<td>Versioning and API Stability</td>
<td>249</td>
</tr>
</tbody>
</table>

Python Module Index | 251
This documentation contains everything you need to know about Scrapy.
CHAPTER 1

Getting help

Having trouble? We’d like to help!

- 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.
- Search for information in the archives of the scrapy-users mailing list, or post a question.
- Ask a question in the #scrapy IRC channel,
- Report bugs with Scrapy in our issue tracker.
Scrapy at a glance

Scrapy is an application framework for crawling web sites and extracting structured data which can be used for a wide range of useful applications, like data mining, information processing or historical archival.

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.

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:

```python
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:
    next_page = response.urljoin(next_page)
    yield scrapy.Request(next_page, callback=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):

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

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.
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!

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!

Installation guide

Installing Scrapy

Scrapy runs on Python 2.7 and Python 3.3 or above (except on Windows where Python 3 is not supported yet).

If you’re already familiar with installation of Python packages, you can install Scrapy and its dependencies from PyPI with:

```
pip install Scrapy
```
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.

### Platform specific installation notes

**Windows**

- Install Python 2.7 from [https://www.python.org/downloads/](https://www.python.org/downloads/)

  You need to adjust PATH environment variable to include paths to the Python executable and additional scripts. The following paths need to be added to PATH:

  ```
  C:\Python27;C:\Python27\Scripts;
  ```

  To update the PATH open a Command prompt and run:

  ```
  c:\python27\python.exe c:\python27\tools\scripts\win_add2path.py
  ```

  Close the command prompt window and reopen it so changes take effect, run the following command and check it shows the expected Python version:

  ```
  python --version
  ```

- Install pywin32 from [http://sourceforge.net/projects/pywin32/](http://sourceforge.net/projects/pywin32/)

  Be sure you download the architecture (win32 or amd64) that matches your system


  Now open a Command prompt to check pip is installed correctly:

  ```
  pip --version
  ```

- At this point Python 2.7 and pip package manager must be working, let’s install Scrapy:

  ```
  pip install Scrapy
  ```

**Note:** Python 3 is not supported on Windows. This is because Scrapy core requirement Twisted does not support Python 3 on Windows.

**Ubuntu 12.04 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 12.04, 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 Wheezy (7.0) 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 http://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

**Anaconda**

Using Anaconda is an alternative to using a virtualenv and installing with pip.

*Note:* For Windows users, or if you have issues installing through pip, this is the recommended way to install Scrapy.

If you already have Anaconda or Miniconda installed, the conda-forge community have up-to-date packages for Linux, Windows and OS X.

To install Scrapy using conda, run:

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

**Scrapy Tutorial**

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

We are going to scrape 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, you may find useful the online book *Learn Python The Hard Way*. You can also take a look at this list of Python resources for non-programmers.

**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:

```bash
scrapy startproject tutorial
```

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

```python
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 http://quotes.toscrape.com domain. You will get an output similar to this:

```
2016-09-20 14:48:00 [scrapy] INFO: Spider opened
2016-09-20 14:48:00 [scrapy] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items_ (at 0 items/min)
2016-09-20 14:48:00 [scrapy] DEBUG: Telnet console listening on 127.0.0.1:6023
2016-09-20 14:48:00 [scrapy] DEBUG: Crawled (200) <GET http://quotes.toscrape.com/page/1/> (referer: None)
2016-09-20 14:48:01 [quotes] DEBUG: Saved file quotes-1.html
2016-09-20 14:48:01 [scrapy] 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:

```python
import scrapy

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

2.3. Scrapy Tutorial
Scrapy Documentation, Release 1.2.3

```python
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 ... ]
[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='&lt;title&gt;Quotes to Scrape&lt;/title&gt;'>]
```

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:
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:

```python
>>> 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:

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

As an alternative, you could've written:

```python
>>> 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:

```python
>>> 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 or extensions like Firebug (see sections about Using Firebug for scraping and Using Firefox 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:

```python
>>> 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:

```html
<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>
  by <small class="author">Albert Einstein</small>
  <a href="/author/Albert-Einstein">(about)</a>
</div>
```

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

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

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

```python
>>> 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:

```python
>>> 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:

```python
>>> 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:
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:

```python
>>> 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
```
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 used 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.

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:

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

We can try extracting it in the shell:

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

```python
>>> 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):
    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:
            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.

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)').extract():
            yield scrapy.Request(response.urljoin(href),
                                    callback=self.parse_author)

        # follow pagination links
        next_page = response.css('li.next a::attr(href)').extract_first()
        if next_page is not None:
            next_page = response.urljoin(next_page)
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.

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 `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.

### 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:

```python
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('span small a::text').extract_first(),
}

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, 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.

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 at a glance 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.

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 at a glance   Understand what Scrapy is and how it can help you.
Installation guide   Get Scrapy installed on your computer.
Scrapy Tutorial      Write your first Scrapy project.
Examples             Learn more by playing with a pre-made Scrapy project.
CHAPTER 3

Basic concepts

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.)

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)
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
    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
```

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.

### 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 `myproject`.

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` 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 his spiders.

3.1. Command line tool 27
Usage example:

```bash
$ 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.

Usage examples:

```bash
$ 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.

Usage example:

```bash
$ 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.

Usage example:

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

**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
- **--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 ... ]
```

**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.*

**Custom project commands**

You can also add your custom project commands by using the `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:
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:

```python
from setuptools import setup, find_packages

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

## Spiders

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

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

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

   The first requests to perform are obtained by calling the `start_requests()` method which (by default) generates `Request` for the URLs specified in the `start_urls` and the `parse` method as callback function for the Requests.

2. In the callback function, you parse the response (web page) and return either dicts with extracted data, `Item` objects, `Request` objects, or an iterable of these objects. Those Requests will also contain a callback (maybe the same) and will then be downloaded by Scrapy and then their response handled by the specified callback.

3. In callback functions, you parse the page contents, typically using `Selectors` (but you can also use BeautifulSoup, lxml or whatever mechanism you prefer) and generate items with the parsed data.

4. Finally, the items returned from the spider will be typically persisted to a database (in some `Item Pipeline`) or written to a file using `Feed exports`.

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

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

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

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

Note: In Python 2 this must be ASCII only.

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

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

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

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

crawler
This attribute is set by the from_crawler() class method after 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.

This is the method called by Scrapy when the spider is opened for scraping when no particular URLs are specified. If particular URLs are specified, the **make_requests_from_url()** is used instead to create the Requests. This method is also called only once from Scrapy, so it’s safe to implement it as a generator.

The default implementation uses **make_requests_from_url()** to generate Requests 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:

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

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

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

**make_requests_from_url(url)**

A method that receives a URL and returns a **Request** object (or a list of **Request** objects) to scrape. This method is used to construct the initial requests in the **start_requests()** method, and is typically used to convert urls to requests.

Unless overridden, this method returns Requests with the **parse()** method as their callback function, and with dont_filter parameter enabled (see **Request** class for more info).

**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**.

3.2. Spiders
closed(reason)
Called when the spider closes. This method provides a shortcut to signals.connect() for the spider_closed signal.

Let's see an example:

```python
import scrapy

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

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

        Return multiple Requests and items from a single callback:

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

            def parse(self, response):
                for h3 in response.xpath('//h3').extract():
                    yield {'title': h3}
                for url in response.xpath('//a/@href').extract():
                    yield scrapy.Request(url, callback=self.parse)

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

        ```
        import scrapy
        from myproject.items import MyItem

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

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

            def parse(self, response):
                for h3 in response.xpath('//h3').extract():
                    yield MyItem(title=h3)
```
for url in response.xpath('//a/@href').extract():
    yield scrapy.Request(url, callback=self.parse)

**Spider arguments**

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

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

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

Spiders can access arguments in their `__init__` methods:

```python
import scrapy
class MySpider(scrapy.Spider):
    name = 'myspider'
    def __init__(self, category=None, *args, **kwargs):
        super(MySpider, self).__init__(*args, **kwargs)
        self.start_urls = ['http://www.example.com/categories/%s' % category]
```

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

**Generic Spiders**

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

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

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

**CrawlSpider**

```python
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:

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

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

    rules = (  # Extract links matching 'category.php' (but not matching 'subsection.php')
        # and follow links from them (since no callback means follow=True by default).
```

```
Rule(LinkExtractor(allow=("category\.php", ), deny=("subsection\.php", )))

    # Extract links matching 'item\.php' and parse them with the spider's method
    def parse_item(self, response):
        self.logger.info('Hi, this is an item page! \$s', response.url)
        item = scrapy.Item()
        item['id'] = response.xpath('//td[@id="item_id"]//text()').re(r'ID: \(d+\)')
        item['name'] = response.xpath('//td[@id="item_name"]//text()').extract()
        item['description'] = response.xpath('//td[@id="item_description"]//text()').extract()
        return item

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

**XMLFeedSpider**

class scrapy.spiders.XMLFeedSpider

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

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

**iterator**

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

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

It defaults to: 'iternodes'.

**itertag**

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

    itertag = 'product'

**namespaces**

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

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

Example:
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, your 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:

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

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

    def parse_node(self, response, node):
        self.logger.info('Hi, this is a <{0}> node!: {1}', 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 rows contained in the file CSV feed which will be used to extract fields from it.

**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`:

```python
from scrapy.spiders import CSVFeedSpider
from myproject.items import TestItem
class MySpider(CSVFeedSpider):
    name = 'example.com'
    allowed_domains = ['example.com']
    start_urls = ['http://www.example.com/feed.csv']
delimiter = ','
quotechar = '"'
human = ['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:
Scrapy Documentation, Release 1.2.3

• **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:

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

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

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

**sitemap_follow**

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

By default, all sitemaps are followed.

**sitemap_alternate_links**

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

For example:

```xml
<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:

```python
from scrapy.spiders import SitemapSpider

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

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

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

```python
from scrapy.spiders import SitemapSpider

class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/sitemap.xml']
    sitemap_rules = [('/product/', 'parse_product'),
                     ('/category/', 'parse_category'),
                     ]
```
Follow sitemaps defined in the robots.txt file and only follow sitemaps whose url contains /sitemap_shop:

```python
from scrapy.spiders import SitemapSpider
class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]
    sitemap_follow = ['/sitemap_shops']
    def parse_shop(self, response):
        pass  # ... scrape shop here ...
```

Combine SitemapSpider with other sources of urls:

```python
from scrapy.spiders import SitemapSpider
class MySpider(SitemapSpider):
    sitemap_urls = ['http://www.example.com/robots.txt']
    sitemap_rules = [
        ('/shop/', 'parse_shop'),
    ]
    other_urls = ['http://www.example.com/about']
    def start_requests(self):
        requests = list(super(MySpider, self).start_requests())
        requests += [scrapy.Request(x, self.parse_other) for x in self.other_urls]
        return requests
    def parse_shop(self, response):
        pass  # ... scrape shop here ...
    def parse_other(self, response):
        pass  # ... scrape other here ...
```

### 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

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:

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

Constructing from text:

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

Constructing from response:

```python
>>> 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:

```python
>>> 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:


Here’s its HTML code:

```html
<html>
<head>
  <base href='http://example.com/' />
  <title>Example website</title>
</head>
<body>
  <div id='images'>
```
First, let’s open the shell:

```
```

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`:

```
3.3. Selectors 43
```
Notice that CSS selectors can select text or attribute nodes using CSS3 pseudo-elements:

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

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

```python
>>> 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',
 'image2.html',
 'image3.html',
 'image4.html',
 'image5.html']

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

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

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

```python
>>> links = response.xpath('//a[contains(@href, "image")]

>>> links.extract()
[u'<a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>',
 'a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>',
 'a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>',
 'a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>',
 'a href="image1.html">Name: My image 1 <br><img src="image1_thumb.jpg"></a>']
```
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:

```python
>>> 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:

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

Working with relative XPaths

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:

```python
>>> 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:

```python
>>> 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 .// XPath):

```python
>>> for p in divs.xpath('.//p'):  # extracts all <p> inside
    print p.extract()
```
Another common case would be to extract all direct `<p>` children:

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

For more details about relative XPaths see the Location Paths section in the XPath specification.

### 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:

<table>
<thead>
<tr>
<th>prefix</th>
<th>namespace</th>
<th>usage</th>
</tr>
</thead>
<tbody>
<tr>
<td>re</td>
<td><a href="http://exslt.org/regular-expressions">http://exslt.org/regular-expressions</a></td>
<td>regular expressions</td>
</tr>
<tr>
<td>set</td>
<td><a href="http://exslt.org/sets">http://exslt.org/sets</a></td>
<td>set manipulation</td>
</tr>
</tbody>
</table>

### 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:

```python
>>> 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](http://schema.org/Product)) with groups of itemscopes and corresponding itemprops:

```python
>>> 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//@id').extract()
[u'id1', u'id2', u'id3', u'id4', u'id5']
>>> sel.xpath('//li[@id="id\d"]').extract()
[u'link1.html', u'link2.html', u'link4.html', u'link5.html']
```
>>> doc = ""
... <div itemprop itemtype="http://schema.org/Product">
...   <span itemprop="name">Kenmore White 17" Microwave</span>
... <img src="kenmore-microwave-17in.jpg" alt='Kenmore 17" Microwave' />
...   <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" />
...   </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>/5 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>/5 stars
...     </div>
...     <span itemprop="description">Great microwave for the price. It is small and
...       fits in my apartment.</span>
...   </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(./*@itemprop,
...       ./*[@itemscope]/@itemprop''')
...   print "properties:", props.extract()
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:

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

Converting a node-set to string:

```python
>>> sel.xpath('//a//text()').extract() # take a peek at the node-set
[u'Click here to go to the ']

>>> sel.xpath("string(//a[1]//text())").extract() # convert it to string
[u'Click here to go to the ']
```
Scrapy Documentation, Release 1.2.3

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:

3.3. Selectors

49


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:

```python
>>> 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.

Built-in Selectors reference

```python
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.

```python
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)`.

**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 built-in `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:

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

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

```python
sel.xpath('//h1')
```

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

```python
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:

```python
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:

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

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

```python
sel.xpath('//product')
```

2. Extract all prices from a Google Base XML feed which requires registering a namespace:

```python
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:
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):

```python
>>> response.xpath('//link')
[]
```

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

```python
>>> 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 manu-
ally, 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.

## 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 finding memory leaks (see Debugging memory leaks with `trackref`), etc.

### Declaring Items

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

```python
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.

**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.

**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**

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

**Getting field values**

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

>>> product['price']
1000

>>> product['last_updated']
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):
```
```python
...

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

```python

```python
>>> 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:

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

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

Other common tasks

Copying items:

```python
>>> 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:
Creating items from dicts:

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

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

## 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:

```python
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:

```python
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.

### Item objects

```python
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.

### Field objects

```python
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.
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.

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:

```python
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.

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):

```python
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.

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.

### Declaring Item Loaders

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

```python
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.

### 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:

```python
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(
        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'&euro;', 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.
**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:

```python
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):

   ```python
   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):

   ```python
   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:

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

**ItemLoader objects**

```python
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:

```python
>>> 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:

```python
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

- **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:

```python
# 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`.

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

**Parameters**

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

**Examples:**

```python
# 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:**

```python
# 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:**

```python
# 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.

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:

```html
<footer>
    <a class="social" href="http://facebook.com/whatever">Like Us</a>
    <a class="social" href="http://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:
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:

```python
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()
```

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.

### 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):

```python
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:

```python
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.

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()
>>> 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:

```python
>>> 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:

```python
>>> 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'scrapy'])
[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:
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.

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:

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

Launch the shell

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

```
scrapy shell <url>
```
Scrapy Documentation, Release 1.2.3

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:

```plaintext
# 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.

### 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(request_or_url)` - fetch a new response from the given request or URL 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

Example of shell session

Here’s an example of a typical shell session where we start by scraping the http://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 'http://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]   crawler <scrapy.crawler.Crawler object at 0x1e16b50>
[s]   item {}
[s]   request <GET http://scrapy.org>
[s]   response <200 http://scrapy.org>
[s]   settings <scrapy.settings.Settings object at 0x2bfd650>
[s]   spider <Spider 'default' at 0x20c6f50>
[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
```

After that, we can start playing with the objects:

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

>>> fetch("http://reddit.com")
```

```
[s] Available Scrapy objects:
[s]   crawler <scrapy.crawler.Crawler object at 0x7fb3ed9c9c90>
[s]   item {}
[s]   request <GET http://reddit.com>
[s]   response <200 https://www.reddit.com/>
[s]   settings <scrapy.settings.Settings object at 0x7fb3ed9c10>
[s]   spider <DefaultSpider 'default' at 0x7fb3ed5390>
[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
```

```
>>> response.xpath('//title/text()').extract()
[u'reddit: the front page of the internet']
```
>>> request = request.replace(method="POST")

>>> fetch(request)

Available Scrapy objects:

crawler   <scrapy.crawler.Crawler object at 0x1e16b50>
...

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

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:

```python
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:

```
Available Scrapy objects:
crawler   <scrapy.crawler.Crawler object at 0x1e16b50>
...
>>> response.url
'http://example.org'
```

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

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

```
>>> ^D
...
```

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.

## 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

### Writing your own item pipeline

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

```python
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:

```python
open_spider(self, spider)
```

This method is called when the spider is opened.

**Parameters**

- `spider` (`Spider` object) – the spider which was opened

```python
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

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:

```python
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.jl file, containing one item per line serialized in JSON format:

```python
import json
class JsonWriterPipeline(object):
    def open_spider(self, spider):
        self.file = open('items.jl', 'wb')
    def close_spider(self, spider):
        self.file.close()
    def process_item(self, item, spider):
        line = json.dumps(dict(item)) + '
        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:

```python
import pymongo

class MongoPipeline(object):
    collection_name = 'scrapy_items'

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

```python
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, callback=self.screenshot)

        return item
```

3.7. Item Pipeline
def 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)
    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:

```python
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
```

Activating an Item Pipeline component

To activate an Item Pipeline component you must add its class to the `ITEM_PIPELINES` setting, like in the following example:

```python
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.
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.

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 FEED_EXPORTERS setting.

JSON

- \texttt{FEED\_FORMAT}: json
- Exporter used: JsonItemExporter
- See \textit{this warning} if you’re using JSON with large feeds.

JSON lines

- \texttt{FEED\_FORMAT}: jsonlines
- Exporter used: JsonLinesItemExporter

CSV

- \texttt{FEED\_FORMAT}: csv
- Exporter used: CsvItemExporter
- To specify columns to export and their order use \texttt{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

- \texttt{FEED\_FORMAT}: xml
- Exporter used: XmlItemExporter
Pickle

- `FEED_FORMAT`: pickle
- Exporter used: `PickleItemExporter`

Marshal

- `FEED_FORMAT`: marshal
- Exporter used: `MarshalItemExporter`

Storages

When using the feed exports you define where to store the feed using a URI (through the `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).

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`

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

  • AWS_ACCESS_KEY_ID
  • 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

Settings

These are the settings used for configuring the feed exports:

  • FEED_URI (mandatory)
  • FEED_FORMAT
  • FEED_STORAGES
  • FEED_EXPORTERS
  • FEED_STORE_EMPTY
  • FEED_EXPORT_ENCODING
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_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:

```python
{
    '': '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 `FEED_STORAGES`. E.g., to disable the built-in FTP storage backend (without replacement), place this in your `settings.py`:

```python
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:

```python
{
    '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 `FEED_EXPORTERS`. E.g., to disable the built-in CSV exporter (without replacement), place this in your `settings.py`:

```python
FEED_EXPORTERS = {
    'csv': None,
}
```

**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.

### Request objects

**class** `scrapy.http.Request(url[, callback, method='GET', headers, body, cookies, meta, encoding='utf-8', priority=0, dont_filter=False, errback])`

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:

   ```python
   request_with_cookies = Request(url="http://www.example.com",
                                  cookies={'currency': 'USD',
                                           'country': 'UY'})
   ```

2. Using a list of dicts:

   ```python
   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:

```python
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.

**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.

3.9. Requests and Responses 81
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:

```python
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:

```python
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
    return request

def parse_page2(self, response):
    item = response.meta['item']
    item['other_url'] = response.url
    return 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 raise 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:

```python
import scrapy

from scrapy.spidermiddlewares.httperror import HttpError
from twisted.internet.error import DNSLookupError, TimeoutError, TCPTimedOutError

class ErrbackSpider(scrapy.Spider):
    name = "errback_example"
    start_urls = ["http://www.httpbin.org/",]  # HTTP 200 expected
```
```python
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)
```

### 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:

- `dont_redirect`
- `dont_retry`
- `handle_httpstatus_list`
- `handle_httpstatus_all`
- `dont_merge_cookies` (see `cookies` parameter of `Request` constructor)
- `cookiejar`
- `dont_cache`
redirect_urls
bindaddress
dont_obey_robotstxt
download_timeout
download_maxsize
download_latency
proxy

**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: `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.

**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.

```python
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:

```python
classmethod from_response(response[, formname=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.

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.
- **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.
- **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.

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...

Response objects

class scrapy.http.Response(url='[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** *(str)* – the response body. It must be str, not unicode, unless you’re using 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:
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 Down- loader Middlewares (although you have the Request available there by other means) and handlers of the 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:

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

```python
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

```python
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`.

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 import 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.

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`:

```python
from scrapy.linkextractors import LinkExtractor
```

There used to be other link extractor classes in previous Scrapy versions, but they are deprecated now.

LxmlLinkExtractor

```python
class scrapy.linkextractors.lxmlhtml.LxmlLinkExtractor(allow=(), deny=(), allow_domains=(), deny_domains=(), deny_extensions=None, restrict_xpaths=(), restrict_css=(), tags=('a', 'area'), attrs=('href',), canonicalize=True, unique=True, process_value=None)
```

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 True.

• **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**:

```python
def process_value(value):
    m = re.search("javascript:goToPage\('(.+)\'\", value)
    if m:
        return m.group(1)
```

**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.

**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.

**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:

```python
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.

How to access settings

In a spider, the settings are available through self.settings:

```python
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:

```python
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.

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.

**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.

**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 AUTOTHROTTLE_TARGET_CONCURRENCY option.
CONCURRENT_REQUESTS_PER_IP

Default: 0

The maximum number of concurrent (i.e. simultaneous) requests that will be performed to any single IP. If non-zero, the 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 DOWNLOAD_DELAY and AutoThrottle extension: if 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 Scrapy shell.

DEFAULT_REQUEST_HEADERS

Default:

```python
{
    '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 REDIRECT_PRIORITY_ADJUST and RETRY_PRIORITY_ADJUST.
**DEPTH_STATS**

Default: True
Scope: scrapy.spidermiddlewares.depth.DepthMiddleware
Whether to collect maximum depth stats.

**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 `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 `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:
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 `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 `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 * DOWNLOAD_DELAY` and `1.5 * DOWNLOAD_DELAY`.

When `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 `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 `DOWNLOAD_HANDLERS` instead.

You can disable any of these download handlers by assigning `None` to their URI scheme in `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 `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 `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_warnsize` spider attribute and per-request using `download_warnsize` Request.meta key.

This feature needs Twisted >= 11.1.

### 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).

### DUPEFILTER_DEBUG

Default: `False`

By default, RFPDupeFilter only logs the first duplicate request. Setting `DUPEFILTER_DEBUG` to `True` will make it log all duplicate requests.

### EDITOR

Default: `depends on the environment`

The editor to use for editing spiders with the `edit` command. It defaults to the EDITOR environment variable, if set. Otherwise, it defaults to `vi` (on Unix systems) or the IDLE editor (on Windows).

### EXTENSIONS

Default:: `{}`

A dict containing the extensions enabled in your project, and their orders.

### EXTENSIONS_BASE

Default:

```python
{
    '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.

**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:

```python
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 `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 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.

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: False

Scope: scrapy.extensions.memusage

Whether to enable the memory usage extension that will shutdown the Scrapy process when it exceeds a memory limit, and also notify by email when that happened.

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 MEMUSAGE_LIMIT_MB and 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_REPORT**

Default: False  
Scope: scrapy.extensions.memusage  
Whether to send a memory usage report after each spider has been closed.  
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 * `DOWNLOAD_DELAY` and 1.5 * `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 `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] 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 SPIDER_CONTRACTS instead. For more info see Spiders Contracts.
You can disable any of these contracts by assigning None to their class path in 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_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:

```python
{
    '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:

```python
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: [http://www.boutell.com/newfaq/misc/urllength.html](http://www.boutell.com/newfaq/misc/urllength.html)

**USER_AGENT**

Default: "Scrapy/VERSION (+http://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.

- *AJAXCRAWL_ENABLED*
- *AUTOTHROTTLE_DEBUG*
- *AUTOTHROTTLE_ENABLED*
- *AUTOTHROTTLE_MAX_DELAY*
- *AUTOTHROTTLE_START_DELAY*
- *AUTOTHROTTLE_TARGET_CONCURRENCY*
- *AWS_ACCESS_KEY_ID*
- *AWS_SECRET_ACCESS_KEY*
- *BOT_NAME*
- *CLOSESPIDER_ERRORCOUNT*
- *CLOSESPIDER_ITEMCOUNT*
- *CLOSESPIDER_PAGECOUNT*
- *CLOSESPIDER_TIMEOUT*
scrapy documentation, release 1.2.3

• commands_module
• compression_enabled
• concurrent_items
• concurrent_requests
• concurrent_requests_per_domain
• concurrent_requests_per_ip
• cookies_debug
• cookies_enabled
• default_item_class
• default_request_headers
• depth_limit
• depth_priority
• depth_stats
• depth_stats_verbose
• dncache_enabled
• dncache_size
• dns_timeout
• downloader
• downloader_clientcontextfactory
• downloader_client_tls_method
• downloader_httpclientfactory
• downloader_middlewares
• downloader_middlewares_base
• downloader_stats
• download_delay
• download_handlers
• download_handlers_base
• download_maxsize
• download_timeout
• download_warnsize
• dupefilter_class
• dupefilter_debug
• editor
• extensions
• extensions_base
• feed_exporters
• FEED_EXPORTERS_BASE
• FEED_EXPORT_ENCODING
• FEED_EXPORT_FIELDS
• FEED_FORMAT
• FEED_STORAGES
• FEED_STORAGES_BASE
• FEED_STORE_EMPTY
• FEED_TEMPDIR
• FEED_URI
• FILES_EXPIRES
• FILES_RESULT_FIELD
• FILES_STORE
• FILES_STORE_S3_ACL
• FILES_URLS_FIELD
• HTTPCACHE_ALWAYS_STORE
• HTTPCACHE_DBM_MODULE
• HTTPCACHE_DIR
• HTTPCACHE_ENABLED
• HTTPCACHE_EXPIRATION_SECS
• HTTPCACHE_GZIP
• HTTPCACHE_IGNORE_HTTP_CODES
• HTTPCACHE_IGNORE_MISSING
• HTTPCACHE_IGNORE_RESPONSE_CACHE_CONTROLS
• HTTPCACHE_IGNORE_SCHEMES
• HTTPCACHE_POLICY
• HTTPCACHE_STORAGE
• HTTPERROR_ALLOWED_CODES
• HTTPERROR_ALLOW_ALL
• HTTPERROR_AUTH_ENCODING
• IMAGES_EXPIRES
• IMAGES_MIN_HEIGHT
• IMAGES_MIN_WIDTH
• IMAGES_RESULT_FIELD
• IMAGES_STORE
• IMAGES_STORE_S3_ACL
• IMAGES_THUMBS
• IMAGES_URLS_FIELD
• ITEM_PIPELINES
• ITEM_PIPELINES_BASE
• LOG_DATEFORMAT
• LOG_ENABLED
• LOGENCODING
• LOG_FILE
• LOG_FORMAT
• LOG_LEVEL
• LOG_STDOUT
• MAIL_FROM
• MAIL_HOST
• MAIL_PASS
• MAIL_PORT
• MAIL_SSL
• MAIL_TLS
• MAIL_USER
• MEMDEBUG_ENABLED
• MEMDEBUG_NOTIFY
• MEMUSAGE_CHECK_INTERVAL_SECONDS
• MEMUSAGE_ENABLED
• MEMUSAGE_LIMIT_MB
• MEMUSAGE_NOTIFY_MAIL
• MEMUSAGE_REPORT
• MEMUSAGE_WARNING_MB
• METAREFRESH_ENABLED
• METAREFRESH_MAXDELAY
• NEWSPIDER_MODULE
• RANDOMIZE_DOWNLOAD_DELAY
• REACTOR_THREADPOOL_MAXSIZE
• REDIRECT_ENABLED
• REDIRECT_MAX_TIMES
• REDIRECT_MAX_TIMES
• REDIRECT_PRIORITY_ADJUST
• REFERER_ENABLED
• RETRY_ENABLED
• \texttt{RETRY\_HTTP\_CODES}
• \texttt{RETRY\_PRIORITY\_ADJUST}
• \texttt{RETRY\_TIMES}
• \texttt{ROBOTSTXT\_OBEY}
• \texttt{SCHEDULER}
• \texttt{SCHEDULER\_DEBUG}
• \texttt{SCHEDULER\_DISK\_QUEUE}
• \texttt{SCHEDULER\_MEMORY\_QUEUE}
• \texttt{SCHEDULER\_PRIORITY\_QUEUE}
• \texttt{SPIDER\_CONTRACTS}
• \texttt{SPIDER\_CONTRACTS\_BASE}
• \texttt{SPIDER\_LOADER\_CLASS}
• \texttt{SPIDER\_MIDDLEWARES}
• \texttt{SPIDER\_MIDDLEWARES\_BASE}
• \texttt{SPIDER\_MODULES}
• \texttt{STATSMAILER\_RCPTS}
• \texttt{STATS\_CLASS}
• \texttt{STATS\_DUMP}
• \texttt{TELNETCONSOLE\_ENABLED}
• \texttt{TELNETCONSOLE\_HOST}
• \texttt{TELNETCONSOLE\_PORT}
• \texttt{TELNETCONSOLE\_PORT}
• \texttt{TEMPLATES\_DIR}
• \texttt{URLLENGTH\_LIMIT}
• \texttt{USER\_AGENT}

\section*{Exceptions}

\subsection*{Built-in Exceptions reference}

Here’s a list of all exceptions included in Scrapy and their usage.

\textbf{DropItem}

\begin{verbatim}
exception scrapy.exceptions.DropItem
\end{verbatim}

The exception that must be raised by item pipeline stages to stop processing an Item. For more information see \textit{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')

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.
Built-in services

CHAPTER 4

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 built-in 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.

Log levels

Python’s built-in logging defines 5 different levels to indicate severity on 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)

How to log messages

Here’s a quick example of how to log a message using the logging.WARNING level:
Scrapy Documentation, Release 1.2.3

```
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 you need so, last example could be rewrote 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 it’s to create different loggers for every module). These loggers can be configured independently, and they allow hierarchical constructions.

Last 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 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

Logging from Spiders

Scrapy provides a `logger` within each Spider instance, that can be accessed and used like this:

```
import scrapy

class MySpider(scrapy.Spider):
    name = 'myspider'
    start_urls = ['http://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 = ['http://scrapinghub.com']

    def parse(self, response):
        logger.info('Parse function called on %s', response.url)

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:

• LOG_FILE
• LOG_ENABLED
• LOG_ENCODING
• LOG_LEVEL
• LOG_FORMAT
• LOG_DATEFORMAT
• LOG_STDOUT

The first couple of settings define a destination for log messages. If LOG_FILE is set, messages sent through the root logger will be redirected to a file named LOG_FILE with encoding LOG_ENCODING. If unset and LOG_ENABLED is True, log messages will be displayed on the standard error. Lastly, if LOG_ENABLED is False, there won’t be any visible log output.

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.

LOG_FORMAT and 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.

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 LOG_FILE
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.

### Common Stats Collector uses

Access the stats collector through the `stats` attribute. Here is an example of an extension that access stats:

```python
class ExtensionThatAccessStats(object):
    def __init__(self, stats):
        self.stats = stats

    @classmethod
    def from_crawler(cls, crawler):
        return cls(crawler.stats)
```

Set stat value:

```python
stats.set_value('hostname', socket.gethostname())
```

Increment stat value:

```python
stats.inc_value('custom_count')
```

Set stat value only if greater than previous:

```python
stats.max_value('max_items_scraped', value)
```

Set stat value only if lower than previous:

```python
stats.min_value('min_free_memory_percent', value)
```

Get stat value:

```python
>>> stats.get_value('custom_count')
1
```

Get all stats:

```python
>>> stats.get_stats()
{'custom_count': 1, 'start_time': datetime.datetime(2009, 7, 14, 21, 47, 28, 977139)}
```

### 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 `STATS_CLASS` setting. The default Stats Collector used is the MemoryStatsCollector.

4.2. Stats Collection 119
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 `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.

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.

Quick example

There are two ways to instantiate the mail sender. You can instantiate it using the standard constructor:

```python
from scrapy.mail import MailSender
mailer = MailSender()
```

Or you can instantiate it passing a Scrapy settings object, which will respect the `settings`:

```python
mailer = MailSender.from_settings(settings)
```

And here is how to use it to send an e-mail (without attachments):

```python
mailer.send(to=["someone@example.com"], subject="Some subject", body="Some body", cc=["another@example.com"])
```

**Note:** As shown in the example above, `to` and `cc` need to be lists of email addresses, not single addresses, and even for one recipient, i.e. `to="someone@example.com"` will not work.

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, smtpass=None, smtpport=None*)

**Parameters**

- `smtphost (str)` – the SMTP host to use for sending the emails. If omitted, the `MAIL_HOST` setting will be used.
- `mailfrom (str)` – the address used to send emails (in the From: header). If omitted, the `MAIL_FROM` setting will be used.
- `smtpuser` – the SMTP user. If omitted, the `MAIL_USER` setting will be used. If not given, no SMTP authentication will be performed.
- `smtpass (str)` – the SMTP pass for authentication.
- `smtpport (int)` – the SMTP port to connect to
- `smtptls (boolean)` – enforce using SMTP STARTTLS
- `smtpssl (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 (list)` – the e-mail recipients
- `subject (str)` – the subject of the e-mail
- `cc (list)` – 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

**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 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

---

**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.

**How to access the telnet console**

The telnet console listens in the TCP port defined in the TELNETCONSOLE_PORT setting, which defaults to 6023. To access the console you need to type:
You need the telnet program which comes installed by default in Windows, and most Linux distros.

**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:

<table>
<thead>
<tr>
<th>Shortcut</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>crawler</td>
<td>the Scrapy Crawler (scrapy.crawler.Crawler object)</td>
</tr>
<tr>
<td>engine</td>
<td>Crawler.engine attribute</td>
</tr>
<tr>
<td>spider</td>
<td>the active spider</td>
</tr>
<tr>
<td>slot</td>
<td>the engine slot</td>
</tr>
<tr>
<td>extensions</td>
<td>the Extension Manager (Crawler.extensions attribute)</td>
</tr>
<tr>
<td>stats</td>
<td>the Stats Collector (Crawler.stats attribute)</td>
</tr>
<tr>
<td>settings</td>
<td>the Scrapy settings object (Crawler.settings attribute)</td>
</tr>
<tr>
<td>est</td>
<td>print a report of the engine status</td>
</tr>
<tr>
<td>prefs</td>
<td>for memory debugging (see Debugging memory leaks)</td>
</tr>
<tr>
<td>p</td>
<td>a shortcut to the pprint.pprint function</td>
</tr>
<tr>
<td>hpy</td>
<td>for memory debugging (see Debugging memory leaks)</td>
</tr>
</tbody>
</table>

**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.scraper.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.scraper.slot.queue) : 0
  len(engine.scraper.slot.active) : 0
  engine.scraper.slot.active_size : 0
  engine.scraper.slot.itemproc_size : 0
  engine.scraper.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.

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

Telnet settings

These are the settings that control the telnet console’s behaviour:

TELENETCONSOLE_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.

TELENETCONSOLE_HOST

Default: '127.0.0.1'

The interface the telnet console should listen on

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.
Frequently Asked Questions

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.

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:

```python
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):
        # 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.

What Python versions does Scrapy support?

Scrapy is supported under Python 2.7 and Python 3.3+. Python 2.6 support was dropped starting at Scrapy 0.20. Python 3 support was added in Scrapy 1.1.

Note: Python 3 is not yet supported on Windows.

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!

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.

How can I scrape an item with attributes in different pages?

See Passing additional data to callback functions.

Scrapy crashes with: ImportError: No module named win32api

You need to install pywin32 because of this Twisted bug.

How can I simulate a user login in my spider?

See Using FormRequest.from_response() to simulate a user login.
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'
```

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](#).

How can I make Scrapy consume less memory?

See previous question.

Can I use Basic HTTP Authentication in my spiders?

Yes, see [HttpAuthMiddleware](#).

Why does Scrapy download pages in English instead of my native language?

Try changing the default `Accept-Language` request header by overriding the `DEFAULT_REQUEST_HEADERS` setting.

Where can I find some example Scrapy projects?

See [Examples](#).

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.

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](#).
What is the recommended way to deploy a Scrapy crawler in production?

See *Deploying Spiders*.

Can I use JSON for large exports?

It’ll depend on how large your output is. See this warning in *JsonItemExporter* documentation.

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.

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:

```python
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 *DOWNLOAD_DELAY* setting.

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*.

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*.
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.

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 \texttt{xmliter} and \texttt{csviter} from \texttt{scrapy.utils.iterators} module. In fact, this is what the feed spiders (see \texttt{Spiders}) use under the cover.

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 \texttt{Requests and Responses} and \texttt{CookiesMiddleware}.

How can I see the cookies being sent and received from Scrapy?

Enable the \texttt{COOKIES_DEBUG} setting.

How can I instruct a spider to stop itself?

Raise the \texttt{CloseSpider} exception from a callback. For more info see: \texttt{CloseSpider}.

How can I prevent my Scrapy bot from getting banned?

See \texttt{Avoiding getting banned}.

Should I use spider arguments or settings to configure my spider?

Both \texttt{spider arguments} and \texttt{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.

I’m scraping a XML document and my XPath selector doesn’t return any items

You may need to remove namespaces. See \texttt{Removing namespaces}.
Scrapy Documentation, Release 1.2.3

Debugging Spiders

This document explains the most common techniques for debugging spiders. Consider the following scrapy spider below:

```python
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.

### 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=myspider -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=myspider -c parse_item -d 2 -v <item_url>
[ ... scrapy log lines crawling example.com spider ... ]

>>> STATUS DEPTH LEVEL 2 <<<
# Scraped Items

[{'url': <item_url>)]

# Requests

[]
```

Chapter 5. Solving specific problems
Checking items scraped from a single start_url, can also be easily achieved using:

```bash
$ scrapy parse --spider=myspider -d 3 'http://example.com/page1'
```

### 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*):

```python
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.*

### 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:

```python
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.
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:

```python
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.

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:

```python
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:

```python
@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:

```python
@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:
Use the `check` command to run the contract checks.

**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 `SPIDER_CONTRACTS` setting:

```python
SPIDER_CONTRACTS = {
    'myproject.contracts.ResponseCheck': 10,
    'myproject.contracts.ItemValidate': 10,
}
```

Each contract must inherit from `scrapy.contracts.Contract` and can override three methods:

```python
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)
```

```python
adjust_request_args (args)
This receives a dict as an argument containing default arguments for `Request` object. Must return the same or a modified version of it.
```

```python
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.
```

```python
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:

```python
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')
```

5.3. Spiders Contracts
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.

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.

```python
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.

```python
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.

```python
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: %(asctime)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.

**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:

```python
import scrapy
from scrapy.crawler import CrawlerProcess

class MySpider1(scrapy.Spider):
    # Your first spider definition
    ...

class MySpider2(scrapy.Spider):
    # 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`:

```python
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
    ...
```
```python
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:

```python
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.

**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 Scrapyd 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:
Then you fire a spider run on 3 different Scrapyd servers. The spider would receive a (spider) argument `part` with the number of the partition to crawl:

```bash
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
```

### 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 `COOKIES_ENABLED`) as some sites may use cookies to spot bot behaviour
- use download delays (2 or higher). See `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.

### 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 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

5.5. Broad Crawls
• 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.

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
```

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
```

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.

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 preforming large broad crawls in production. Using DEBUG level when developing your (broad) crawler may fine though.

To set the log level use:

```
LOG_LEVEL = 'INFO'
```
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
```

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
```

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
```

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
```

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.

**Using Firefox for scraping**

Here is a list of tips and advice on using Firefox for scraping, along with a list of useful Firefox add-ons to ease the scraping process.

**Caveats with inspecting the live browser DOM**

Since Firefox add-ons 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 when working with Firefox and XPath:

- Disable Firefox Javascript while inspecting the DOM looking for XPaths to be used in Scrapy
- 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.

**Useful Firefox add-ons for scraping**

**Firebug**

Firebug is a widely known tool among web developers and it’s also very useful for scraping. In particular, its Inspect Element feature comes very handy when you need to construct the XPaths for extracting data because it allows you to view the HTML code of each page element while moving your mouse over it.

See *Using Firebug for scraping* for a detailed guide on how to use Firebug with Scrapy.

**XPather**

XPather allows you to test XPath expressions directly on the pages.

**XPath Checker**

XPath Checker is another Firefox add-on for testing XPaths on your pages.

**Tamper Data**

Tamper Data is a Firefox add-on which allows you to view and modify the HTTP request headers sent by Firefox. Firebug also allows to view HTTP headers, but not to modify them.
Firecookie

Firecookie makes it easier to view and manage cookies. You can use this extension to create a new cookie, delete existing cookies, see a list of cookies for the current site, manage cookies permissions and a lot more.

Using Firebug for scraping

**Note:** Google Directory, the example website used in this guide is no longer available as it has been shut down by Google. The concepts in this guide are still valid though. If you want to update this guide to use a new (working) site, your contribution will be more than welcome!. See *Contributing to Scrapy* for information on how to do so.

Introduction

This document explains how to use Firebug (a Firefox add-on) to make the scraping process easier and more fun. For other useful Firefox add-ons see *Useful Firefox add-ons for scraping*. There are some caveats with using Firefox add-ons to inspect pages, see *Caveats with inspecting the live browser DOM*.

In this example, we’ll show how to use Firebug to scrape data from the Google Directory, which contains the same data as the Open Directory Project used in the tutorial but with a different face.

Firebug comes with a very useful feature called Inspect Element which allows you to inspect the HTML code of the different page elements just by hovering your mouse over them. Otherwise you would have to search for the tags manually through the HTML body which can be a very tedious task.

In the following screenshot you can see the Inspect Element tool in action.
At first sight, we can see that the directory is divided in categories, which are also divided in subcategories. However, it seems that there are more subcategories than the ones being shown in this page, so we’ll keep looking:
As expected, the subcategories contain links to other subcategories, and also links to actual websites, which is the purpose of the directory.

### Getting links to follow

By looking at the category URLs we can see they share a pattern:

```
http://directory.google.com/Category/Subcategory/Another_Subcategory
```

Once we know that, we are able to construct a regular expression to follow those links. For example, the following one:

```
directory\..*/[A-Z][a-zA-Z_\-]+\$
```
So, based on that regular expression we can create the first crawling rule:

```python
Rule(LinkExtractor(allow='directory.google.com/[A-Z][a-zA-Z_/]+$',
    'parse_category',
    follow=True,
),
)
```

The `Rule` object instructs `CrawlSpider` based spiders how to follow the category links. `parse_category` will be a method of the spider which will process and extract data from those pages.

This is how the spider would look so far:

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

class GoogleDirectorySpider(CrawlSpider):
    name = 'directory.google.com'
    allowed_domains = ['directory.google.com']
    start_urls = ['http://directory.google.com/']

    rules = (  
        Rule(LinkExtractor(allow='directory\.google\.com/[A-Z][a-zA-Z_/]+$',
            'parse_category', follow=True,
        ),
    )

    def parse_category(self, response):
        # write the category page data extraction code here
        pass
```

### Extracting the data

Now we’re going to write the code to extract data from those pages.

With the help of Firebug, we’ll take a look at some page containing links to websites (say `http://directory.google.com/Top/Arts/Awards/`) and find out how we can extract those links using `Selectors`. We’ll also use the `Scrapy shell` to test those XPath’s and make sure they work as we expect.
As you can see, the page markup is not very descriptive: the elements don’t contain id, class or any attribute that clearly identifies them, so we’ll use the ranking bars as a reference point to select the data to extract when we construct our XPaths.

After using FireBug, we can see that each link is inside a td tag, which is itself inside a tr tag that also contains the link’s ranking bar (in another td).

5.7. Using Firebug for scraping
So we can select the ranking bar, then find its parent (the tr), and then finally, the link’s td (which contains the data we want to scrape).

This results in the following XPath:

```
//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a
```

It’s important to use the Scrapy shell to test these complex XPath expressions and make sure they work as expected.

Basically, that expression will look for the ranking bar’s td element, and then select any td element who has a descendant a element whose href attribute contains the string #pagerank”

Of course, this is not the only XPath, and maybe not the simpler one to select that data. Another approach could be, for example, to find any font tags that have that grey colour of the links,

Finally, we can write our parse_category() method:

```python
def parse_category(self, response):
    # The path to website links in directory page
    links = response.xpath('//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td/font')

    for link in links:
        item = DirectoryItem()
        item['name'] = link.xpath('a/text()').extract()
        item['url'] = link.xpath('a/@href').extract()
        item['description'] = link.xpath('font[2]/text()').extract()
        yield item
```

Be aware that you may find some elements which appear in Firebug but not in the original HTML, such as the typical case of <tbody> elements.

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.

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 `spider_opened` but not releasing them on `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.

**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:

```python
>>> 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 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:

```python
>>> from scrapy.utils.trackref import get_oldest
```

```python
>>> r = get_oldest('HtmlResponse')
>>> r.url
```

If you want to iterate over all objects, instead of getting the oldest one, you can use the `scrapy.utils.trackref.iter_all()` function:

```python
>>> from scrapy.utils.trackref import iter_all
```

```python
>>> [r.url for r in iter_all('HtmlResponse')]
...
```

**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:

```python
>>> from scrapy.spiders import Spider
```

```python
>>> 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.

**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 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  % Referred Via:
   0      22307  8%  16423880 31%  16423880  31 dict
   1      12285  4%  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  91 list
   8       987   0%    456936  1%  49872920  93 scrapy.http.headers.Headers
   9      1209   0%   456936  1%  49872920  94 _sre.SRE_Pattern

<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      10   0%   196888  1% 14498152  88 ['moduleDict'], ['modules']
```
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.

**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.

**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)

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.
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.

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.

Enabling your Media Pipeline

To enable your media pipeline you must first add it to your project ITEM_PIPELINES setting.

For Images Pipeline, use:

```python
ITEM_PIPELINES = {'scrapy.pipelines.images.ImagesPipeline': 1}
```

For Files Pipeline, use:

```python
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 ITEM_PIPELINES setting.

For the Files Pipeline, set the FILES_STORE setting:

```python
FILES_STORE = '/path/to/valid/dir'
```

For the Images Pipeline, set the IMAGES_STORE setting:
Supported Storage

File system is currently the only officially supported storage, but there is also support for storing files in Amazon S3.

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 `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

`FILES_STORE` and `IMAGES_STORE` can represent an Amazon S3 bucket. Scrapy will automatically upload the files to the bucket.

For example, this is a valid `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 `FILES_STORE_S3_ACL` and `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.

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 `FILES_URLS_FIELD` and/or `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 `IMAGES_URLS_FIELD` and/or `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 `MYPipeline.IMAGES_URLS_FIELD` and your custom settings will be used.

### Additional features

#### File expiration

The Image Pipeline avoids downloading files that were downloaded recently. To adjust this retention delay use the `FILES_EXPIRES` setting (or `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 `IMAGES_THUMBS` to a dictionary where the keys are the thumbnail names and the values are their dimensions.

5.9. Downloading and processing files and images
For example:

```python
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:

```plaintext
<IMAGES_STORE>/thumbs/<size_name>/<image_id>.jpg
```

Where:

- `<size_name>` is the one specified in the `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:

```plaintext
<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 `IMAGES_MIN_HEIGHT` and `IMAGES_MIN_WIDTH` settings.

For example:

```plaintext
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.

**Extending the Media Pipelines**

See here the methods that you can override in your custom Files Pipeline:

```python
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 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:

```python
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:

```python
class scrapy.pipelines.images.ImagesPipeline
```

5.9. Downloading and processing files and images
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.

Custom Images pipeline example

Here is a full example of the Images Pipeline whose methods are exemplified above:

```python
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
```

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)

**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 Scrapy, you can use the scrapyd-deploy tool provided by the scrapyd-client package. Please refer to the scrapyd-deploy documentation for more information.

Scrapy is maintained by some of the Scrapy developers.

**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`.

**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.

**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.

**How it works**

AutoThrottle extension adjusts download delays dynamically to make spider send `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 `CONCURRENT_REQUESTS_PER_DOMAIN` or `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.
**Throttling algorithm**

AutoThrottle algorithm adjusts download delays based on the following rules:

1. spiders always start with a download delay of `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 `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 `DOWNLOAD_DELAY` or greater than `AUTOTHROTTLE_MAX_DELAY`.

**Note:** The AutoThrottle extension honours the standard Scrapy settings for concurrency and delay. This means that it will respect `CONCURRENT_REQUESTS_PER_DOMAIN` and `CONCURRENT_REQUESTS_PER_IP` options and never set a download delay lower than `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.

**Settings**

The settings used to control the AutoThrottle extension are:

- `AUTOTHROTTLE_ENABLED`
- `AUTOTHROTTLE_START_DELAY`
- `AUTOTHROTTLE_MAX_DELAY`
- `AUTOTHROTTLE_DEBUG`
- `CONCURRENT_REQUESTS_PER_DOMAIN`
- `CONCURRENT_REQUESTS_PER_IP`
- `DOWNLOAD_DELAY`

For more information see *How it works*.

**AUTOTHROTTLE_ENABLED**

Default: `False`

Enables the AutoThrottle extension.

**AUTOTHROTTLE_START_DELAY**

Default: `5.0`

The initial download delay (in seconds).
**AUTOTHROTTLE_MAX_DELAY**

Default: 60.0

The maximum download delay (in seconds) to be set in case of high latencies.

**AUTOTHROTTLE_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 `AUTOTHROTTLE_TARGET_CONCURRENCY` value (e.g., 0.5) makes the crawler more conservative and polite.

Note that `CONCURRENT_REQUESTS_PER_DOMAIN` and `CONCURRENT_REQUESTS_PER_IP` options are still respected when AutoThrottle extension is enabled. This means that if `AUTOTHROTTLE_TARGET_CONCURRENCY` is set to a value higher than `CONCURRENT_REQUESTS_PER_DOMAIN` or `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 `AUTOTHROTTLE_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.

**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:

```
2013-05-16 13:08:46-0300 [scrapy] INFO: Scrapy 0.17.0 started (bot: scrapybot)
2013-05-16 13:08:47-0300 [scrapy] INFO: Crawled 0 pages (at 0 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:48-0300 [scrapy] INFO: Crawled 74 pages (at 4440 pages/min), scraped 0 items (at 0 items/min)
2013-05-16 13:08:49-0300 [scrapy] INFO: Crawled 143 pages (at 4140 pages/min), scraped 0 items (at 0 items/min)
```
That tells you that Scrapy is able to crawl about 3900 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.

## 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
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 (i.e. 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.

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
```

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):

```python
def parse_item(self, response):
    # parse item here
    self.state['items_count'] = self.state.get('items_count', 0) + 1
```

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:

```python
def some_callback(self, response):
    somearg = 'test'
    return scrapy.Request('http://www.example.com', callback=lambda r: self.other_
                         -callback(r, somearg))
```
```python
def other_callback(self, response, somearg):
    print "the argument passed is:", somearg
```

But this will:

```python
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 `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 Firefox for scraping**  Learn how to scrape with Firefox and some useful add-ons.

**Using Firebug for scraping**  Learn how to scrape efficiently using Firebug.

**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.
Architecture overview

This document describes the architecture of Scrapy and how its components interact.

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 green 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.
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**.
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*.

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

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.

Activating a downloader middleware

To activate a downloader middleware component, add it to the `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:

```python
DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.CustomDownloaderMiddleware': 543,
}
```

The `DOWNLOADER_MIDDLEWARES` setting is merged with the `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 `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 `DOWNLOADER_MIDDLEWARES_BASE` and enabled by default) you must define it in your project’s `DOWNLOADER_MIDDLEWARES` setting and assign `None` as its value. For example, if you want to disable the user-agent middleware:

```python
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.

### Writing your own downloader middleware

Each middleware component is a Python class that defines one or more of the following methods:

```python
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 are 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

### 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 `DOWNLOADER_MIDDLEWARES_BASE` setting.

**CookiesMiddleware**

```python
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:

- `COOKIES_ENABLED`
- `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 `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:

```python
for i, url in enumerate(urls):
    yield scrapy.Request(url, meta={'cookiejar': i},
                         callback=self.parse_page)
```

Keep in mind that the `cookiejar` meta key is not "sticky". You need to keep passing it along on subsequent requests. For example:

```python
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.

**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 `COOKIES_DEBUG` enabled:

```
2011-04-06 14:35:10-0300 [scrapy] INFO: Spider opened
      Cookie: clientlanguage_nl=en_EN
      Set-Cookie: JSESSIONID=B~FA4DC0C496C8762AE4F1A620EAB34F38; Path=/
      Set-Cookie: ip_isocode=US
      Set-Cookie: clientlanguage_nl=en_EN; Expires=Thu, 07-Apr-2011 21:21:34 GMT;
      Path=/
```

**DefaultHeadersMiddleware**

This middleware sets all default requests headers specified in the `DEFAULT_REQUEST_HEADERS` setting.
**DownloadTimeoutMiddleware**

```python
class scrapy.downloadermiddlewares.downloadtimeout.DownloadTimeoutMiddleware
```

This middleware sets the download timeout for requests specified in the `DOWNLOAD_TIMEOUT` setting or `download_timeout` spider attribute.

**Note:** You can also set download timeout per-request using `download_timeout` Request.meta key; this is supported even when DownloadTimeoutMiddleware is disabled.

**HttpAuthMiddleware**

```python
class scrapy.downloadermiddlewares.httpauth.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:

```python
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**

```python
class scrapy.downloadermiddlewares.httpcache.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 two HTTP cache storage backends:

- `Filesystem storage backend (default)`
- `DBM storage backend`

You can change the HTTP cache storage backend with the `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 `HTTPCACHE_POLICY` setting. Or you can also implement your own policy. You can also avoid caching a response on every policy using `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:

- `HTTPCACHE_POLICY` to `scrapy.extensions.httpcache.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:

- `HTTPCACHE_POLICY` to `scrapy.extensions.httpcache.RFC2616Policy`
Filesysterm storage backend (default)

File system storage backend is available for the HTTP cache middleware.

In order to use this storage backend, set:

- `HTTPCACHE_STORAGE` to `scrapy.extensions.httpcache.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 `HTTPCACHE_DBM_MODULE` setting.

In order to use this storage backend, set:

- `HTTPCACHE_STORAGE` to `scrapy.extensions.httpcache.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 `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, `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.

HttpCompressionMiddleware Settings

COMPRESSION_ENABLED

Default: True
Whether the Compression middleware will be enabled.

ChunkedTransferMiddleware

class scrapy.downloadermiddlewares.chunked.ChunkedTransferMiddleware
   This middleware adds support for chunked transfer encoding

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.

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):
   • REDIRECT_ENABLED
   • 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:
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 follow 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):

- `METAREFRESH_ENABLED`
- `METAREFRESH_MAXDELAY`

This middleware obey `REDIRECT_MAX_TIMES` setting, `dont_redirect` and `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):

- `RETRY_ENABLED`
- `RETRY_TIMES`
- `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.

**RETRY_HTTP_CODES**

Default: `[500, 502, 503, 504, 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 `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 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 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 `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.

---

**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_AUTH_ENCODING**

Default: "latin-1"

The default encoding for proxy authentication on HttpProxyMiddleware.
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.

Activating a spider middleware

To activate a spider middleware component, add it to the 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:

```python
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.CustomSpiderMiddleware': 543,
}
```

The SPIDER_MIDDLEWARES setting is merged with the 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 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 built-in middleware (the ones defined in SPIDER_MIDDLEWARES_BASE, and enabled by default) you must define it in your project SPIDER_MIDDLEWARES setting and assign None as its value. For example, if you want to disable the off-site middleware:

```python
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.

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 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 `Response`, 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

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 `SPIDER_MIDDLEWARES_BASE` setting.

**DepthMiddleware**

`class scrapy.spidermiddlewares.depth.DepthMiddleware`

DepthMiddleware is a scrape middleware used for tracking the depth of each Request inside the site being scraped. It can be used to limit the maximum depth to scrape or things like that.

The `DepthMiddleware` can be configured through the following settings (see the settings documentation for more info):

- `DEPTH_LIMIT` - The maximum depth that will be allowed to crawl for any site. If zero, no limit will be imposed.
- `DEPTH_STATS` - Whether to collect depth stats.
- `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 `HTTPERROR_ALLOWED_CODES` setting.

For example, if you want your spider to handle 404 responses you can do this:

```python
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.
Scrapy Documentation, Release 1.2.3

HttpErrorMiddleware settings

HTTPERROR_ALLOWED_CODES

Default: []

Pass all responses with non-200 status codes contained in this list.

HTTPERROR_ALLOW_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:

```
```

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.spidermiddlewaresreferer.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.
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):

• URLLENGTH_LIMIT - The maximum URL length to allow for crawled URLs.

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.

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 hypothetic extension to handle Google Sitemaps would use settings like GOOGLE-SITEMAP_ENABLED, GOOGLESITEMAP_DEPTH, and so on.

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 EXTENSIONS setting in your Scrapy settings. In 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 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 EXTENSIONS setting is merged with the 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 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.

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 HTTPCACHE_ENABLED setting is set.
Disabling an extension

In order to disable an extension that comes enabled by default (i.e., those included in the `EXTENSIONS_BASE` setting) you must set its order to `None`. For example:

```python
EXTENSIONS = {
    'scrapy.extensions.corestats.CoreStats': None,
}
```

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:

```python
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

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)

---

### 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 `TELNETCONSOLE_ENABLED` setting, and the server will listen in the port specified in `TELNETCONSOLE_PORT`.

**Memory usage extension**

class scrapy.extensions.memusage.MemoryUsage

---

6.4. Extensions
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 (MEMUSAGE_WARNING_MB) and when the maximum value is reached (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 MEMUSAGE_ENABLED setting and can be configured with the following settings:

- MEMUSAGE_LIMIT_MB
- MEMUSAGE_WARNING_MB
- MEMUSAGE_NOTIFY_MAIL
- MEMUSAGE_REPORT
- 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 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:

- CLOSESPIDER_TIMEOUT
- CLOSESPIDER_ITEMCOUNT
- CLOSESPIDER_PAGECOUNT
- 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 closepider_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 if items and those items are passed by the item pipeline, the spider will be closed with the reason `closespider_itemcount`. 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.statsmailer.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 `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).

### Core API

New in version 0.15.

This section documents the Scrapy core API, and it’s intended for developers of extensions and middlewares.

### 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 `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 **craw**l() method.

**craw**l (**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.

```python
class scrapy.crawler.CrawlerRunner (settings=None)
    This is a convenient helper class that keeps track of, manages and runs crawlers inside an already setup Twisted reactor.
    The CrawlerRunner object must be instantiated with a **Settings** object.
    This class shouldn’t be needed (since Scrapy is responsible of using it accordingly) unless writing scripts that manually handle the crawling process. See *Run Scrapy from a script* for an example.

crawl (crawler_or_spidercls, **args**, **kwargs)
    Run a crawler with the provided arguments.
    It will call the given Crawler’s **craw**l() method, while keeping track of it so it can be stopped later.
    If crawler_or_spidercls isn’t a Crawler instance, this method will try to create one using this parameter as the spider class given to it.
    Returns a deferred that is fired when the crawling is finished.

Parameters

- **crawler_or_spidercls** (Crawler instance, Spider subclass or string) – already created crawler, or a spider class or spider’s name inside the project to create it
- **args** (list) – arguments to initialize the spider
- **kwargs** (dict) – keyword arguments to initialize the spider
```

**crawlers**
Set of crawlers started by **craw**l() and managed by this class.

**create_crawler**(crawler_or_spidercls)
Return a Crawler object.

- If crawler_or_spidercls is a Crawler, it is returned as-is.
- If crawler_or_spidercls is a Spider subclass, a new Crawler is constructed for it.
•If `crawler_or_spidercls` is a string, this function finds a spider with this name in a Scrapy project (using spider loader), then creates a Crawler instance for it.

```python
join()

Returns a deferred that is fired when all managed crawlers have completed their executions.
```

```python
stop()

Stops simultaneously all the crawling jobs taking place.

Returns a deferred that is fired when they all have ended.
```

```python
class scrapy.crawler.CrawlerProcess(settings=None)
Bases: scrapy.crawler.CrawlerRunner

A class to run multiple scrapy crawlers in a process simultaneously.

This class extends CrawlerRunner by adding support for starting a Twisted reactor and handling shutdown signals, like the keyboard interrupt command Ctrl-C. It also configures top-level logging.

This utility should be a better fit than CrawlerRunner if you aren’t running another Twisted reactor within your application.

The CrawlerProcess object must be instantiated with a Settings object.

This class shouldn’t be needed (since Scrapy is responsible of using it accordingly) unless writing scripts that manually handle the crawling process. See Run Scrapy from a script for an example.

```python
crawl(crawler_or_spidercls, *args, **kwargs)

Run a crawler with the provided arguments.

It will call the given Crawler’s `crawl()` method, while keeping track of it so it can be stopped later.

If `crawler_or_spidercls` isn’t a Crawler instance, this method will try to create one using this parameter as the spider class given to it.

Returns a deferred that is fired when the crawling is finished.
```

**Parameters**

- `crawler_or_spidercls` (`Crawler` instance, `Spider` subclass or string) – already created crawler, or a spider class or spider’s name inside the project to create it
- `args` (`list`) – arguments to initialize the spider
- `kwargs` (`dict`) – keyword arguments to initialize the spider

```python
crawlers

Set of crawlers started by `crawl()` and managed by this class.
```

```python
create_crawler(crawler_or_spidercls)

Return a Crawler object.

•If `crawler_or_spidercls` is a Crawler, it is returned as-is.
•If `crawler_or_spidercls` is a Spider subclass, a new Crawler is constructed for it.
•If `crawler_or_spidercls` is a string, this function finds a spider with this name in a Scrapy project (using spider loader), then creates a Crawler instance for it.
```

```python
join()

Returns a deferred that is fired when all managed crawlers have completed their executions.
```

```python
start(stop_after_crawl=True)

This method starts a Twisted reactor, adjusts its pool size to `REACTOR_THREADPOOL_MAXSIZE`, and installs a DNS cache based on `DNSCACHE_ENABLED` and `DNSCACHE_SIZE`.
```
If `stop_after_crawl` is True, the reactor will be stopped after all crawlers have finished, using `join()`.

**Parameters**

- `stop_after_crawl (boolean)` – stop or not the reactor when all crawlers have finished

**stop()**

- Stops simultaneously all the crawling jobs taking place.
- Returns a deferred that is fired when they all have ended.

### 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.

```python
SETTINGS_PRIORITIES = {
    'default': 0,
    'command': 10,
    'project': 20,
    'spider': 30,
    'cmdline': 40,
}
```

For a detailed explanation on each settings sources, see: `Settings`.

**scrapy.settings.get_settings_priority (priority)**

Small helper function that looks up a given string priority in the `SETTINGS_PRIORITIES` dictionary and returns its numerical value, or directly returns a given numerical priority.

**class scrapy.settings.Settings (values=None, priority='project')**

Bases: `scrapy.settings.BaseSettings`

This object stores Scrapy settings for the configuration of internal components, and can be used for any further customization.

It is a direct subclass and supports all methods of `BaseSettings`. Additionally, after instantiation of this class, the new object will have the global default settings described on `Built-in settings reference` already populated.

**class scrapy.settings.BaseSettings (values=None, priority='project')**

Instances of this class behave like dictionaries, but store priorities along with their `(key, value)` pairs, and can be frozen (i.e. marked immutable).

Key-value entries can be passed on initialization with the `values` argument, and they would take the priority level (unless `values` is already an instance of `BaseSettings`, in which case the existing priority levels will be kept). If the `priority` argument is a string, the priority name will be looked up in `SETTINGS_PRIORITIES`. Otherwise, a specific integer should be provided.

Once the object is created, new settings can be loaded or updated with the `set()` method, and can be accessed with the square bracket notation of dictionaries, or with the `get()` method of the instance and its value conversion variants. When requesting a stored key, the value with the highest priority will be retrieved.

**copy()**

- Make a deep copy of current settings.

  This method returns a new instance of the `Settings` class, populated with the same values and their priorities.
Modifications to the new object won’t be reflected on the original settings.

copy_to_dict()
Make a copy of current settings and convert to a dict.
This method returns a new dict populated with the same values and their priorities as the current settings.
Modifications to the returned dict won’t be reflected on the original settings.
This method can be useful for example for printing settings in Scrapy shell.

freeze()
Disable further changes to the current settings.
After calling this method, the present state of the settings will become immutable. Trying to change values through the set() method and its variants won’t be possible and will be alerted.

frozencopy()
Return an immutable copy of the current settings.
Alias for a freeze() call in the object returned by copy().

get(name, default=None)
Get a setting value without affecting its original type.

Parameters
• name (string) – the setting name
• default (any) – the value to return if no setting is found

getbool(name, default=False)
Get a setting value as a boolean.
1, '1', and True return True, while 0, '0', False and None return False.
For example, settings populated through environment variables set to '0' will return False when using this method.

Parameters
• name (string) – the setting name
• default (any) – the value to return if no setting is found

getdict(name, default=None)
Get a setting value as a dictionary. If the setting original type is a dictionary, a copy of it will be returned. If it is a string it will be evaluated as a JSON dictionary. In the case that it is a BaseSettings instance itself, it will be converted to a dictionary, containing all its current settings values as they would be returned by get(), and losing all information about priority and mutability.

Parameters
• name (string) – the setting name
• default (any) – the value to return if no setting is found

getfloat(name, default=0.0)
Get a setting value as a float.

Parameters
• name (string) – the setting name
• default (any) – the value to return if no setting is found
getint (name, default=0)
Get a setting value as an int.

Parameters

• name (string) – the setting name
• default (any) – the value to return if no setting is found

getlist (name, default=None)
Get a setting value as a list. If the setting original type is a list, a copy of it will be returned. If it’s a string it will be split by ",".

For example, settings populated through environment variables set to 'one,two' will return a list ['one', 'two'] when using this method.

Parameters

• name (string) – the setting name
• default (any) – the value to return if no setting is found

getpriority (name)
Return the current numerical priority value of a setting, or None if the given name does not exist.

Parameters

name (string) – the setting name

gewithbase (name)
Get a composition of a dictionary-like setting and its _BASE counterpart.

Parameters

name (string) – name of the dictionary-like setting

maxpriority ()
Return the numerical value of the highest priority present throughout all settings, or the numerical value for default from SETTINGS_PRIORITIES if there are no settings stored.

set (name, value, priority='project')
Store a key/value attribute with a given priority.

Settings should be populated before configuring the Crawler object (through the configure() method), otherwise they won’t have any effect.

Parameters

• name (string) – the setting name
• value (any) – the value to associate with the setting
• priority (string or int) – the priority of the setting. Should be a key of SETTINGS_PRIORITIES or an integer

setmodule (module, priority='project')
Store settings from a module with a given priority.

This is a helper function that calls set() for every globally declared uppercase variable of module with the provided priority.

Parameters

• module (module object or string) – the module or the path of the module
• priority (string or int) – the priority of the settings. Should be a key of SETTINGS_PRIORITIES or an integer
update \((values, priority=’project’)\)
Store key/value pairs with a given priority.

This is a helper function that calls `set()` for every item of `values` with the provided `priority`.

If `values` is a string, it is assumed to be JSON-encoded and parsed into a dict with `json.loads()` first. If it is a `BaseSettings` instance, the per-key priorities will be used and the `priority` parameter ignored. This allows inserting/updating settings with different priorities with a single command.

Parameters

- `values` (dict or string or `BaseSettings`) – the settings names and values
- `priority` (string or int) – the priority of the settings. Should be a key of `SETTINGS_PRIORITIES` or an integer

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 `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 in the modules of the `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

Signals API

class `scrapy.signalmanager.SignalManager` (`sender=_Anonymous`)  

`connect` \((receiver, signal, **kwargs)\)
Connect a receiver function to a signal.

The signal can be any object, although Scrapy comes with some predefined signals that are documented in the `Signals` section.

Parameters

- `receiver` (callable) – the function to be connected
• **signal** (*object*) – the signal to connect to

**disconnect** (*receiver, signal, **kwargs*)

Disconnect a receiver function from a signal. This has the opposite effect of the **connect()** method, and the arguments are the same.

**disconnect_all** (*signal, **kwargs*)

Disconnect all receivers from the given signal.

**Parameters**

- **signal** (*object*) – the signal to disconnect from

**send_catch_log** (*signal, **kwargs*)

Send a signal, catch exceptions and log them.

The keyword arguments are passed to the signal handlers (connected through the **connect()** method).

**send_catch_log Deferred** (*signal, **kwargs*)

Like **send_catch_log()** but supports returning **Deferreds** from signal handlers.

Returns a **Deferred** that gets fired once all signal handlers deferreds were fired. Send a signal, catch exceptions and log them.

The keyword arguments are passed to the signal handlers (connected through the **connect()** method).

### 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\( (\text{spider}) \)
- Open the given spider for stats collection.

close_spider\( (\text{spider}) \)
- Close the given spider. After this is called, no more specific stats can be accessed or collected.

## 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:

```python
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
```

### Deferred signal handlers

Some signals support returning Twisted deferreds from their handlers, see the Built-in signals reference below to know which ones.

### 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 spider_opened signal, depending on how the spider was started. So don't rely on this signal getting fired before 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

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 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 `spider_closed` signal is sent.

You can, for example, schedule some requests in your `spider_idle` handler 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

**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

**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
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.

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 an Item Exporter to export scraped items to different files, one per spider:

```python
from scrapy import signals
from scrapy.exporters import XmlItemExporter

class XmlExportPipeline(object):
    def __init__(self):
        self.files = {}  

    @classmethod
    def from_crawler(cls, crawler):
        pipeline = cls()
        crawler.signals.connect(pipeline.spider_opened, signals.spider_opened)
        crawler.signals.connect(pipeline.spider_closed, signals.spider_closed)
        return pipeline

    def spider_opened(self, spider):
        file = open('%s_products.xml' % spider.name, 'w+b')
        self.files[spider] = file
        self.exporter = XmlItemExporter(file)
        self.exporter.start_exporting()

    def spider_closed(self, spider):
        self.exporter.finish_exporting()
        file = self.files.pop(spider)
        file.close()

    def process_item(self, item, spider):
        self.exporter.export_item(item)
        return item
```
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:

```python
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:

```python
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)
```

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:

```python
Item(name='Color TV', price='1200')
Item(name='DVD player', price='200')
```

`BaseItemExporter`

```python
class scrapy.exporters.BaseItemExporter(fields_to_export=None, export_empty_fields=False, encoding='utf-8')
```

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`.

**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* instance) 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.

### XmlItemExporter

**class** `scrapy.exporters.XmlItemExporter` *(file, item_element='item', root_element='items', **kwargs)*
Exports Items in XML format to the specified file object.

**Parameters**
Scrapy documentation, Release 1.2.3

- `file` – the file-like object to use for exporting the data.
- `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
<?xml version="1.0" encoding="utf-8"?>
<items>
    <item>
        <name>Color TV</name>
        <price>1200</price>
    </item>
    <item>
        <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:

```python
Item(name=['John', 'Doe'], age='23')
```

Would be serialized as:

```xml
<?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.
- `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

The `PickleItemExporter` class exports Items in pickle format to the given file-like object.

**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.
- `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

The `PprintItemExporter` class exports Items in pretty print format to the specified file object.

**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.

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

The `JsonItemExporter` class 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.

**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.

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.

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).
CHAPTER 7

All the rest

Release notes

Scrapy 1.2.3 (2017-03-03)

• Packaging fix: disallow unsupported Twisted versions in setup.py

Scrapy 1.2.2 (2016-12-06)

Bug fixes

• Fix a cryptic traceback when a pipeline fails on open_spider() (issue 2011)
• Fix embedded IPython shell variables (fixing issue 396 that re-appeared in 1.2.0, fixed in issue 2418)
• A couple of patches when dealing with robots.txt:
  – handle (non-standard) relative sitemap URLs (issue 2390)
  – handle non-ASCII URLs and User-Agents in Python 2 (issue 2373)

Documentation

• Document "download_latency" key in Request's meta dict (issue 2033)
• Remove page on (deprecated & unsupported) Ubuntu packages from ToC (issue 2335)
• A few fixed typos (issue 2346, issue 2369, issue 2369, issue 2380) and clarifications (issue 2354, issue 2325, issue 2414)
Other changes

• Advertise conda-forge as Scrapy’s official conda channel (issue 2387)
• More helpful error messages when trying to use .css() or .xpath() on non-Text Responses (issue 2264)
  startproject command now generates a sample middlewares.py file (issue 2335)
• Add more dependencies’ version info in scrapy version verbose output (issue 2404)
• Remove all *.pyc files from source distribution (issue 2386)

Scrapy 1.2.1 (2016-10-21)

Bug fixes

• Include OpenSSL’s more permissive default ciphers when establishing TLS/SSL connections (issue 2314).
• Fix “Location” HTTP header decoding on non-ASCII URL redirects (issue 2321).

Documentation

• Fix JsonWriterPipeline example (issue 2302).
• Various notes: issue 2330 on spider names, issue 2329 on middleware methods processing order, issue 2327 on getting multi-valued HTTP headers as lists.

Other changes

• Removed www. from start_urls in built-in spider templates (issue 2299).

Scrapy 1.2.0 (2016-10-03)

New Features

• New FEED_EXPORT_ENCODING setting to customize the encoding used when writing items to a file. This can be used to turn off \uXXXX escapes in JSON output. This is also useful for those wanting something else than UTF-8 for XML or CSV output (issue 2034).
• startproject command now supports an optional destination directory to override the default one based on the project name (issue 2005).
• New SCHEDULER_DEBUG setting to log requests serialization failures (issue 1610).
• JSON encoder now supports serialization of set instances (issue 2058).
• Interpret application/json-amazonui-streaming as TextResponse (issue 1503).
• scrapy is imported by default when using shell tools (shell.inspect_response) (issue 2248).
Bug fixes

- DefaultRequestHeaders middleware now runs before UserAgent middleware (issue 2088). **Warning: this is technically backwards incompatible**, though we consider this a bug fix.
- HTTP cache extension and plugins that use the `.scrapy` data directory now work outside projects (issue 1581). **Warning: this is technically backwards incompatible**, though we consider this a bug fix.
- `Selector` does not allow passing both `response` and `text` anymore (issue 2153).
- Fixed logging of wrong callback name with `scrapy parse` (issue 2169).
- Fix for an odd gzip decompression bug (issue 1606).
- Fix for selected callbacks when using CrawlSpider with `scrapy parse` (issue 2225).
- Fix for invalid JSON and XML files when spider yields no items (issue 872).
- Implement `flush()` for `StreamLogger` avoiding a warning in logs (issue 2125).

Refactoring

- `canonicalize_url` has been moved to `w3lib.url` (issue 2168).

Tests & Requirements

Scrapy’s new requirements baseline is Debian 8 “Jessie”. It was previously Ubuntu 12.04 Precise. What this means in practice is that we run continuous integration tests with these (main) packages versions at a minimum: Twisted 14.0, pyOpenSSL 0.14, lxml 3.4.

Scrapy may very well work with older versions of these packages (the code base still has switches for older Twisted versions for example) but it is not guaranteed (because it’s not tested anymore).

Documentation

- Grammar fixes: issue 2128, issue 1566.
- Download stats badge removed from README (issue 2160).
- New `scrapy architecture diagram` (issue 2165).
- Updated `Response` parameters documentation (issue 2197).
- Reworded misleading `RANDOMIZEDOWNLOAD_DELAY` description (issue 2190).
- Add StackOverflow as a support channel (issue 2257).

Scrapy 1.1.4 (2017-03-03)

- Packaging fix: disallow unsupported Twisted versions in setup.py

Scrapy 1.1.3 (2016-09-22)

Bug fixes

- Class attributes for subclasses of `ImagesPipeline` and `FilesPipeline` work as they did before 1.1.1 (issue 2243, fixes issue 2198)
Scrapy Documentation, Release 1.2.3

Documentation

- **Overview** and tutorial rewritten to use http://toscrape.com websites (issue 2236, issue 2249, issue 2252).

Scrapy 1.1.2 (2016-08-18)

Bug fixes

- Introduce a missing IMAGES_STORE_S3_ACL setting to override the default ACL policy in ImagesPipeline when uploading images to S3 (note that default ACL policy is “private” – instead of “public-read” – since Scrapy 1.1.0)
- IMAGES_EXPIRES default value set back to 90 (the regression was introduced in 1.1.1)

Scrapy 1.1.1 (2016-07-13)

Bug fixes

- Add “Host” header in CONNECT requests to HTTPS proxies (issue 2069)
- Use response body when choosing response class (issue 2001, fixes issue 2000)
- Do not fail on canonicalizing URLs with wrong netlocs (issue 2038, fixes issue 2010)
- a few fixes for HttpCompressionMiddleware (and SitemapSpider):
  - Do not decode HEAD responses (issue 2008, fixes issue 1899)
  - Handle charset parameter in gzip Content-Type header (issue 2050, fixes issue 2049)
  - Do not decompress gzip octet-stream responses (issue 2065, fixes issue 2063)
- Catch (and ignore with a warning) exception when verifying certificate against IP-address hosts (issue 2094, fixes issue 2092)
- Make FilesPipeline and ImagesPipeline backward compatible again regarding the use of legacy class attributes for customization (issue 1989, fixes issue 1985)

New features

- Enable genspider command outside project folder (issue 2052)
- Retry HTTPS CONNECT TunnelError by default (issue 1974)

Documentation

- FEED_TEMPDIR setting at lexicographical position (commit 9b3c72c)
- Use idiomatic .extract_first() in overview (issue 1994)
- Update years in copyright notice (commit c2c8036)
- Add information and example on errbacks (issue 1995)
- Use “url” variable in downloader middleware example (issue 2015)
- Grammar fixes (issue 2054, issue 2120)
• New FAQ entry on using BeautifulSoup in spider callbacks (issue 2048)
• Add notes about scrapy not working on Windows with Python 3 (issue 2060)
• Encourage complete titles in pull requests (issue 2026)

Tests

• Upgrade py.test requirement on Travis CI and Pin pytest-cov to 2.2.1 (issue 2095)

Scrapy 1.1.0 (2016-05-11)

This 1.1 release brings a lot of interesting features and bug fixes:

• Scrapy 1.1 has beta Python 3 support (requires Twisted >= 15.5). See Beta Python 3 Support for more details and some limitations.

• Hot new features:
  – Item loaders now support nested loaders (issue 1467).
  – FormRequest.from_response improvements (issue 1382, issue 1137).
  – Added setting AUTOTHROTTLE_TARGET_CONCURRENCY and improved AutoThrottle docs (issue 1324).
  – Added response.text to get body as unicode (issue 1730).
  – Anonymous S3 connections (issue 1358).
  – Deferreds in downloader middlewares (issue 1473). This enables better robots.txt handling (issue 1471).
  – HTTP caching now follows RFC2616 more closely, added settings HTTPCACHE_ALWAYS_STORE and HTTPCACHE_IGNORE_RESPONSE_CACHE_CONTROLS (issue 1151).
  – Selectors were extracted to the parsel library (issue 1409). This means you can use Scrapy Selectors without Scrapy and also upgrade the selectors engine without needing to upgrade Scrapy.
  – HTTPS downloader now does TLS protocol negotiation by default, instead of forcing TLS 1.0. You can also set the SSL/TLS method using the new DOWNLOADER_CLIENT_TLS_METHOD.

• These bug fixes may require your attention:
  – Don’t retry bad requests (HTTP 400) by default (issue 1289). If you need the old behavior, add 400 to RETRY_HTTP_CODES.
  – Fix shell files argument handling (issue 1710, issue 1550). If you try scrapy shell index.html it will try to load the URL http://index.html, use scrapy shell ./index.html to load a local file.
  – Robots.txt compliance is now enabled by default for newly-created projects (issue 1724). Scrapy will also wait for robots.txt to be downloaded before proceeding with the crawl (issue 1735). If you want to disable this behavior, update ROBOTSTXT_OBEY in settings.py file after creating a new project.
  – Exporters now work on unicode, instead of bytes by default (issue 1080). If you use PythonItemExporter, you may want to update your code to disable binary mode which is now deprecated.
  – Accept XML node names containing dots as valid (issue 1533).
  – When uploading files or images to S3 (with FilesPipeline or ImagesPipeline), the default ACL policy is now “private” instead of “public” Warning: backwards incompatible! You can use FILES_STORE_S3_ACL to change it.
– We’ve reimplemented `canonicalize_url()` for more correct output, especially for URLs with non-ASCII characters (issue 1947). This could change link extractors output compared to previous scrapy versions. This may also invalidate some cache entries you could still have from pre-1.1 runs. **Warning: backwards incompatible!**

Keep reading for more details on other improvements and bug fixes.

### Beta Python 3 Support

We have been hard at work to make Scrapy run on Python 3. As a result, now you can run spiders on Python 3.3, 3.4 and 3.5 (Twisted >= 15.5 required). Some features are still missing (and some may never be ported).

Almost all builtin extensions/middlewares are expected to work. However, we are aware of some limitations in Python 3:

- Scrapy does not work on Windows with Python 3
- Sending emails is not supported
- FTP download handler is not supported
- Telnet console is not supported

### Additional New Features and Enhancements

- Scrapy now has a [Code of Conduct](issue 1681).
- Command line tool now has completion for zsh (issue 934).
- Improvements to `scrapy shell`:
  - Support for `bpython` and configure preferred Python shell via `SCRAPY_PYTHON_SHELL` (issue 1100, issue 1444).
  - Support URLs without scheme (issue 1498) **Warning: backwards incompatible!**
  - Bring back support for relative file path (issue 1710, issue 1550).
- Added `MEMUSAGE_CHECK_INTERVAL_SECONDS` setting to change default check interval (issue 1282).
- Download handlers are now lazy-loaded on first request using their scheme (issue 1390, issue 1421).
- HTTPS download handlers do not force TLS 1.0 anymore; instead, OpenSSL’s `SSLv23_method()` / `TLS_method()` is used allowing to try negotiating with the remote hosts the highest TLS protocol version it can (issue 1794, issue 1629).
- `RedirectMiddleware` now skips the status codes from `handle_httpstatus_list` on spider attribute or in `Request`’s `meta` key (issue 1334, issue 1364, issue 1447).
- Form submission:
  - now works with `<button>` elements too (issue 1469).
  - an empty string is now used for submit buttons without a value (issue 1472)
- Dict-like settings now have per-key priorities (issue 1135, issue 1149 and issue 1586).
- Sending non-ASCII emails (issue 1662)
- `CloseSpider` and `SpiderState` extensions now get disabled if no relevant setting is set (issue 1723, issue 1725).
- Added method `ExecutionEngine.close` (issue 1423).
• Added method `CrawlerRunner.create_crawler` (issue 1528).
• Scheduler priority queue can now be customized via `SCHEDULER_PRIORITY_QUEUE` (issue 1822).
• `.pps` links are now ignored by default in link extractors (issue 1835).
• temporary data folder for FTP and S3 feed storages can be customized using a new `FEED_TEMPDIR` setting (issue 1847).
• `FilesPipeline` and `ImagesPipeline` settings are now instance attributes instead of class attributes, enabling spider-specific behaviors (issue 1891).
• `JsonItemExporter` now formats opening and closing square brackets on their own line (first and last lines of output file) (issue 1950).
• If available, `botocore` is used for `S3FeedStorage`, `S3DownloadHandler` and `S3FilesStore` (issue 1761, issue 1883).
• Tons of documentation updates and related fixes (issue 1291, issue 1302, issue 1335, issue 1683, issue 1660, issue 1642, issue 1721, issue 1727, issue 1879).
• Other refactoring, optimizations and cleanup (issue 1476, issue 1481, issue 1477, issue 1315, issue 1290, issue 1750, issue 1881).

**Deprecations and Removals**

• Added `to_bytes` and `to_unicode`, deprecated `str_to_unicode` and `unicode_to_str` functions (issue 778).
• `binary_is_text` is introduced, to replace use of `isbinarytext` (but with inverse return value) (issue 1851)
• The `optional_features` set has been removed (issue 1359).
• The `--lsprof` command line option has been removed (issue 1689). **Warning: backward incompatible**, but doesn’t break user code.
• The following datatypes were deprecated (issue 1720):
  – `scrapy.utils.datatypes.MultiValueDictKeyError`  
  – `scrapy.utils.datatypes.MultiValueDict`  
  – `scrapy.utils.datatypes.SiteNode`  
• The previously bundled `scrapy.xlib.pydispatcher` library was deprecated and replaced by `pydispatcher`.

**Relocations**

• `telnetconsole` was relocated to `extensions/` (issue 1524).
  – Note: `telnet` is not enabled on Python 3 (https://github.com/scrapy/scrapy/pull/1524#issuecomment-146985595)

**Bugfixes**

• Scrapy does not retry requests that got a HTTP 400 Bad Request response anymore (issue 1289). **Warning: backwards incompatible**!
• Support empty password for `http_proxy` config (issue 1274).
**Scrapy Documentation, Release 1.2.3**

- Interpret `application/x-json` as `TextResponse` (issue 1333).
- Support link rel attribute with multiple values (issue 1201).
- Fixed `scrapy.http.FormRequest.from_response` when there is a `<base>` tag (issue 1564).
- Fixed `TEMPLATES_DIR` handling (issue 1575).
- Various `FormRequest` fixes (issue 1595, issue 1596, issue 1597).
- Makes `_monkeypatches` more robust (issue 1634).
- Fixed bug on `XMLItemExporter` with non-string fields in items (issue 1738).
- Fixed startproject command in OS X (issue 1635).
- Fixed PythonItemExporter and CSVExporter for non-string item types (issue 1737).
- Various logging related fixes (issue 1294, issue 1419, issue 1263, issue 1624, issue 1654, issue 1722, issue 1726 and issue 1303).
- Fixed bug in `utils.template.render_templatefile()` (issue 1212).
- Sitemaps extraction from `robots.txt` is now case-insensitive (issue 1902).
- HTTPS+CONNECT tunnels could get mixed up when using multiple proxies to same remote host (issue 1912).

**Scrapy 1.0.7 (2017-03-03)**

- Packaging fix: disallow unsupported Twisted versions in setup.py

**Scrapy 1.0.6 (2016-05-04)**

- FIX: RetryMiddleware is now robust to non-standard HTTP status codes (issue 1857)
- FIX: Filestorage HTTP cache was checking wrong modified time (issue 1875)
- DOC: Support for Sphinx 1.4+ (issue 1893)
- DOC: Consistency in selectors examples (issue 1869)

**Scrapy 1.0.5 (2016-02-04)**

- FIX: [Backport] Ignore bogus links in LinkExtractors (fixes issue 907, commit 108195e)
- TST: Changed buildbot makefile to use 'pytest' (commit 1f3d90a)
- DOC: Fixed typos in tutorial and media-pipeline (commit 808a9ea and commit 803bd87)
- DOC: Add AjaxCrawlMiddleware to `DOWNLOADERS` in settings docs (commit aa94121)

**Scrapy 1.0.4 (2015-12-30)**

- Ignoring xlib/tx folder, depending on Twisted version. (commit 7dfa979)
- Run on new travis-ci infra (commit 6e42f0b)
- Spelling fixes (commit 823a1cc)
- escape nodename in xmliter regex (commit da3c155)
• test xml nodename with dots (commit 4418fc3)
• TST don’t use broken Pillow version in tests (commit a55078c)
• disable log on version command. closes #1426 (commit 86fc330)
• disable log on startproject command (commit db4c9fe)
• Add PyPI download stats badge (commit df2b944)
• don’t run tests twice on Travis if a PR is made from a scrapy/scrapy branch (commit a83ab41)
• Add Python 3 porting status badge to the README (commit 73ac80d)
• fixed RFPDupeFilter persistence (commit 97d080c)
• TST a test to show that dupefilter persistence is not working (commit 97f2fb3)
• explicit close file on file:// scheme handler (commit d9b4850)
• Disable dupefilter in shell (commit c0d0734)
• DOC: Add captions to toctrees which appear in sidebar (commit aa239ad)
• DOC Removed pywin32 from install instructions as it’s already declared as dependency. (commit 10eb400)
• Added installation notes about using Conda for Windows and other OSes. (commit 1c3600a)
• Fixed minor grammar issues. (commit 7f4ddd5)
• fixed a typo in the documentation. (commit b71f677)
• Version 1 now exists (commit 5456c0e)
• fix another invalid xpath error (commit 0a1366e)
• fix ValueError: Invalid XPath: //div[@id="not-exists"]/text() on selectors.rst (commit ca8d60f)
• Typos corrections (commit 7067117)
• fix typos in downloader-middleware.rst and exceptions.rst, middleware -> middleware (commit 32f115c)
• Add note to ubuntu install section about debian compatibility (commit 23fda69)
• Replace alternative OSX install workaround with virtualenv (commit 98b63ee)
• Reference Homebrew’s homepage for installation instructions (commit 1925db1)
• Add oldest supported tox version to contributing docs (commit 5d10d6d)
• Note in install docs about pip being already included in python>=2.7.9 (commit 85c980e)
• Add non-python dependencies to Ubuntu install section in the docs (commit fbd010d)
• Add OS X installation section to docs (commit d8f4eba)
• DOC(ENH): specify path to rtd theme explicitly (commit de73b1a)
• minor: scrapy.Spider docs grammar (commit 1ddcc7b)
• Make common practices sample code match the comments (commit 1b85bdf)
• nextcall repetitive calls (heartbeats). (commit 55f7104)
• Backport fix compatibility with Twisted 15.4.0 (commit b262411)
• pin pytest to 2.7.3 (commit a635c2)
• Merge pull request #1512 from mgedmin/patch-1 (commit 8876111)
• Merge pull request #1513 from mgedmin/patch-2 (commit 5d4daf8)
Scrapy Documentation, Release 1.2.3

- Typo (commit f8d0682)
- Fix list formatting (commit 5f83a93)
- fix scrapy squeue tests after recent changes to queuelib (commit 3365c01)
- Merge pull request #1475 from rweindl/patch-1 (commit 2d688cd)
- Update tutorial.rst (commit fbc1f25)
- Merge pull request #1449 from rhoekman/patch-1 (commit 7d6538c)
- Small grammatical change (commit 8752294)
- Add openssl version to version command (commit 13c45ac)

**Scrapy 1.0.3 (2015-08-11)**

- add service_identity to scrapy install_requires (commit cbc2501)
- Workaround for travis#296 (commit 66af9cd)

**Scrapy 1.0.2 (2015-08-06)**

- Twisted 15.3.0 does not raises PicklingError serializing lambda functions (commit b04dd7d)
- Minor method name fix (commit 6f85c7f)
- minor: scrapy.Spider grammar and clarity (commit 9e9d2e0)
- Put a blurb about support channels in CONTRIBUTING (commit c63882b)
- Fixed typos (commit a9ae7b0)
- Fix doc reference. (commit 7c8a4fe)

**Scrapy 1.0.1 (2015-07-01)**

- Unquote request path before passing to FTPClient, it already escape paths (commit cc00ad2)
- include tests/ to source distribution in MANIFEST.in (commit eca227e)
- DOC Fix SelectJmes documentation (commit b8567bc)
- DOC Bring Ubuntu and Archlinux outside of Windows subsection (commit 392233f)
- DOC remove version suffix from ubuntu package (commit 5303e66)
- DOC Update release date for 1.0 (commit c89fa29)

**Scrapy 1.0.0 (2015-06-19)**

You will find a lot of new features and bugfixes in this major release. Make sure to check our updated overview to get a glance of some of the changes, along with our brushed tutorial.
Support for returning dictionaries in spiders

Declaring and returning Scrapy Items is no longer necessary to collect the scraped data from your spider, you can now return explicit dictionaries instead.

Classic version

```python
class MyItem(scrapy.Item):
    url = scrapy.Field()

class MySpider(scrapy.Spider):
    def parse(self, response):
        return MyItem(url=response.url)
```

New version

```python
class MySpider(scrapy.Spider):
    def parse(self, response):
        return {'url': response.url}
```

Per-spider settings (GSoC 2014)

Last Google Summer of Code project accomplished an important redesign of the mechanism used for populating settings, introducing explicit priorities to override any given setting. As an extension of that goal, we included a new level of priority for settings that act exclusively for a single spider, allowing them to redefine project settings.

Start using it by defining a `custom_settings` class variable in your spider:

```python
class MySpider(scrapy.Spider):
    custom_settings = {
        "DOWNLOAD_DELAY": 5.0,
        "RETRY_ENABLED": False,
    }
```

Read more about settings population: Settings

Python Logging

Scrapy 1.0 has moved away from Twisted logging to support Python built in's as default logging system. We're maintaining backward compatibility for most of the old custom interface to call logging functions, but you'll get warnings to switch to the Python logging API entirely.

Old version

```python
from scrapy import log
log.msg('MESSAGE', log.INFO)
```

New version

```python
import logging
logging.info('MESSAGE')
```

Logging with spiders remains the same, but on top of the `log()` method you'll have access to a custom `logger` created for the spider to issue log events:
class MySpider(scrapy.Spider):
    def parse(self, response):
        self.logger.info('Response received')

Read more in the logging documentation: Logging

Crawler API refactoring (GSOC 2014)

Another milestone for last Google Summer of Code was a refactoring of the internal API, seeking a simpler and easier usage. Check new core interface in: Core API

A common situation where you will face these changes is while running Scrapy from scripts. Here’s a quick example of how to run a Spider manually with the new API:

```
from scrapy.crawler import CrawlerProcess

process = CrawlerProcess({
    'USER_AGENT': 'Mozilla/4.0 (compatible; MSIE 7.0; Windows NT 5.1)'
})
process.crawl(MySpider)
process.start()
```

Bear in mind this feature is still under development and its API may change until it reaches a stable status.

See more examples for scripts running Scrapy: Common Practices

Module Relocations

There’s been a large rearrangement of modules trying to improve the general structure of Scrapy. Main changes were separating various subpackages into new projects and dissolving both scrapy.contrib and scrapy.contrib.ex into top level packages. Backward compatibility was kept among internal relocations, while importing deprecated modules expect warnings indicating their new place.

Full list of relocations

Outsourced packages

Note: These extensions went through some minor changes, e.g. some setting names were changed. Please check the documentation in each new repository to get familiar with the new usage.

<table>
<thead>
<tr>
<th>Old location</th>
<th>New location</th>
</tr>
</thead>
<tbody>
<tr>
<td>scrapy.commands.deploy</td>
<td>scrapyd-client (See other alternatives here: Deploying Spiders)</td>
</tr>
<tr>
<td>scrapy.contrib.djangoitem</td>
<td>scrapy-djangoitem</td>
</tr>
<tr>
<td>scrapy.webservice</td>
<td>scrapy-jsonrpc</td>
</tr>
</tbody>
</table>

scrapy.contrib.exp and scrapy.contrib dissolutions
Old location                                      New location                                      
scrapy.contrib_exp.downloadermiddleware.decompression  scrapy.downloadermiddlewares.decompression 
scrapy.contrib_exp.iterators                          scrapy.utils.iterators                           
scrapy.contrib.downloadermiddleware                    scrapy.downloadermiddlewares                    
scrapy.contrib.exporter                                scrapy.exporters                                
scrapy.contrib.linkextractors                          scrapy.linkextractors                           
scrapy.contrib.loader                                  scrapy.loader                                   
scrapy.contrib.loader.processor                        scrapy.loader.processors                         
scrapy.contrib.pipeline                                scrapy.pipelines                                
scrapy.contrib.spidermiddleware                        scrapy.spidermiddlewares                        
scrapy.contrib.spiders                                  scrapy.spiders                                  
  • scrapy.contrib.closespider                          scrapy.extensions.*                             
  • scrapy.contrib.corestats                           
  • scrapy.contrib.debug                                
  • scrapy.contrib.feedexport                          
  • scrapy.contrib.httpcache                           
  • scrapy.contrib.logstats                            
  • scrapy.contrib.memdebug                            
  • scrapy.contrib.menususage                          
  • scrapy.contrib.spiderstate                         
  • scrapy.contrib.statsmailer                         
  • scrapy.contrib.throttle                            

Plural renames and Modules unification

<table>
<thead>
<tr>
<th>Old location</th>
<th>New location</th>
</tr>
</thead>
<tbody>
<tr>
<td>scrapy.command</td>
<td>scrapy.commands</td>
</tr>
<tr>
<td>scrapy.dupefilter</td>
<td>scrapy.dupefilters</td>
</tr>
<tr>
<td>scrapy.linkextractor</td>
<td>scrapy.linkextractors</td>
</tr>
<tr>
<td>scrapy.spider</td>
<td>scrapy.spiders</td>
</tr>
<tr>
<td>scrapy.squeue</td>
<td>scrapy.squeues</td>
</tr>
<tr>
<td>scrapy.statscol</td>
<td>scrapy.statscollectors</td>
</tr>
<tr>
<td>scrapy.utils.decorator</td>
<td>scrapy.utils.decorators</td>
</tr>
</tbody>
</table>

Class renames

<table>
<thead>
<tr>
<th>Old location</th>
<th>New location</th>
</tr>
</thead>
<tbody>
<tr>
<td>scrapy.spidermanager.SpiderManager</td>
<td>scrapy.spiderloader.SpiderLoader</td>
</tr>
</tbody>
</table>

Settings renames

<table>
<thead>
<tr>
<th>Old location</th>
<th>New location</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPIDER_MANAGER_CLASS</td>
<td>SPIDER_LOADER_CLASS</td>
</tr>
</tbody>
</table>

Changelog

New Features and Enhancements

- Python logging (issue 1060, issue 1235, issue 1236, issue 1240, issue 1259, issue 1278, issue 1286)
- FEED_EXPORT_FIELDS option (issue 1159, issue 1224)
- Dns cache size and timeout options (issue 1132)
- support namespace prefix in xmliter_lxml (issue 963)
- Reactor threadpool max size setting (issue 1123)
• Allow spiders to return dicts. (issue 1081)
• Add Response.urljoin() helper (issue 1086)
• look in ~/.config/scrapy.cfg for user config (issue 1098)
• handle TLS SNI (issue 1101)
• Selectorlist extract first (issue 624, issue 1145)
• Added JmesSelect (issue 1016)
• add gzip compression to filesystem http cache backend (issue 1020)
• CSS support in link extractors (issue 983)
• httpcache dont_cache meta #19 #689 (issue 821)
• add signal to be sent when request is dropped by the scheduler (issue 961)
• avoid download large response (issue 946)
• Allow to specify the quotecchar in CSVFeedSpider (issue 882)
• Add referer to “Spider error processing” log message (issue 795)
• process robots.txt once (issue 896)
• GSoC Per-spider settings (issue 854)
• Add project name validation (issue 817)
• GSoC API cleanup (issue 816, issue 1128, issue 1147, issue 1148, issue 1156, issue 1185, issue 1187, issue 1258, issue 1268, issue 1276, issue 1285, issue 1284)
• Be more responsive with IO operations (issue 1074 and issue 1075)
• Do leveldb compaction for httpcache on closing (issue 1297)

Deprecations and Removals

• Deprecate htmlparser link extractor (issue 1205)
• remove deprecated code from FeedExporter (issue 1155)
• a leftover for.15 compatibility (issue 925)
• drop support for CONCURRENT_REQUESTS_PER_SPIDER (issue 895)
• Drop old engine code (issue 911)
• Deprecate SgmlLinkExtractor (issue 777)

Relocations

• Move exporters/__init__.py to exporters.py (issue 1242)
• Move base classes to their packages (issue 1218, issue 1233)
• Module relocation (issue 1181, issue 1210)
• rename SpiderManager to SpiderLoader (issue 1166)
• Remove djangoitem (issue 1177)
• remove scrapy deploy command (issue 1102)
• dissolve contrib_exp (issue 1134)
• Deleted bin folder from root, fixes #913 (issue 914)
• Remove jsonrpc based webservice (issue 859)
• Move Test cases under project root dir (issue 827, issue 841)
• Fix backward incompatibility for relocated paths in settings (issue 1267)

Documentation

• CrawlerProcess documentation (issue 1190)
• Favoring web scraping over screen scraping in the descriptions (issue 1188)
• Some improvements for Scrapy tutorial (issue 1180)
• Documenting Files Pipeline together with Images Pipeline (issue 1150)
• deployment docs tweaks (issue 1164)
• Added deployment section covering scrapy-deploy and shub (issue 1124)
• Adding more settings to project template (issue 1073)
• some improvements to overview page (issue 1106)
• Updated link in docs/topics/architecture.rst (issue 647)
• DOC reorder topics (issue 1022)
• updating list of Request.meta special keys (issue 1071)
• DOC document download_timeout (issue 898)
• DOC simplify extension docs (issue 893)
• Leaks docs (issue 894)
• DOC document from_crawler method for item pipelines (issue 904)
• Spider_error doesn’t support deferreds (issue 1292)
• Corrections & Sphinx related fixes (issue 1220, issue 1219, issue 1196, issue 1172, issue 1171, issue 1169, issue 1160, issue 1154, issue 1127, issue 1112, issue 1105, issue 1041, issue 1082, issue 1033, issue 944, issue 866, issue 864, issue 796, issue 1260, issue 1271, issue 1293, issue 1298)

Bugfixes

• Item multi inheritance fix (issue 353, issue 1228)
• ItemLoader.load_item: iterate over copy of fields (issue 722)
• Fix Unhandled error in Deferred (RobotsTxtMiddleware) (issue 1131, issue 1197)
• Force to read DOWNLOAD_TIMEOUT as int (issue 954)
• scrapy.utils.misc.load_object should print full traceback (issue 902)
• Fix bug for ".local” host name (issue 878)
• Fix for Enabled extensions, middlewares, pipelines info not printed anymore (issue 879)
• fix dont_merge_cookies bad behaviour when set to false on meta (issue 846)

Python 3 In Progress Support

• disable scrapy.telnet if twisted.conch is not available (issue 1161)
• fix Python 3 syntax errors in ajaxcrawl.py (issue 1162)
• more python3 compatibility changes for urllib (issue 1121)
• assertItemsEqual was renamed to assertCountEqual in Python 3. (issue 1070)
• Import unittest.mock if available. (issue 1066)
• updated deprecated cgi.parse_qsl to use six’s parse_qsl (issue 909)
• Prevent Python 3 port regressions (issue 830)
• PY3: use MutableMapping for python 3 (issue 810)
• PY3: use six.BytesIO and six.moves.cStringIO (issue 803)
• PY3: fix xmlrpclib and email imports (issue 801)
• PY3: use six for robotparser and urlparse (issue 800)
• PY3: use six.iterkeys, six.iteritems, and tempfile (issue 799)
• PY3: fix has_key and use six.moves.configparser (issue 798)
• PY3: use six.moves.cPickle (issue 797)
• PY3 make it possible to run some tests in Python3 (issue 776)

Tests
• remove unnecessary lines from py3-ignores (issue 1243)
• Fix remaining warnings from pytest while collecting tests (issue 1206)
• Add docs build to travis (issue 1234)
• TST don’t collect tests from deprecated modules. (issue 1165)
• install service_identity package in tests to prevent warnings (issue 1168)
• Fix deprecated settings API in tests (issue 1152)
• Add test for webclient with POST method and no body given (issue 1089)
• py3-ignores.txt supports comments (issue 1044)
• modernize some of the asserts (issue 835)
• selector.__repr__ test (issue 779)

Code refactoring
• CSVFeedSpider cleanup: use iterate_spider_output (issue 1079)
• remove unnecessary check from scrapy.utils.spider.iter_spider_output (issue 1078)
• Pydispatch pep8 (issue 992)
• Removed unused ‘load=False’ parameter from walk_modules() (issue 871)
• For consistency, use job_dir helper in SpiderState extension. (issue 805)
• rename “sflo” local variables to less cryptic “log_observer” (issue 775)

Scrapy 0.24.6 (2015-04-20)
• encode invalid xpath with unicode_escape under PY2 (commit 07cb3e5)
• fix IPython shell scope issue and load IPython user config (commit 2c8e573)
• Fix small typo in the docs (commit d694019)
• Fix small typo (commit f92fa83)
• Converted sel.xpath() calls to response.xpath() in Extracting the data (commit c2c6d15)
Scrappy 0.24.5 (2015-02-25)

- Support new _getEndpoint Agent signatures on Twisted 15.0.0 (commit 540b9bc)
- DOC a couple more references are fixed (commit b4c454b)
- DOC fix a reference (commit e3c1260)
- t.i.b.ThreadedResolver is now a new-style class (commit 9e13f42)
- S3DownloadHandler: fix auth for requests with quoted paths/query params (commit cdb9a0b)
- fixed the variable types in mailsender documentation (commit bb3a848)
- Reset items_scraped instead of item_count (commit edb07a4)
- Tentative attention message about what document to read for contributions (commit 7ee6f7a)
- mitmproxy 0.10.1 needs netlib 0.10.1 too (commit 874fcdd)
- pin mitmproxy 0.10.1 as >0.11 does not work with tests (commit c6b21f0)
- Test the parse command locally instead of against an external url (commit c3a6628)
- Patches Twisted issue while closing the connection pool on HTTPDownloadHandler (commit d0bf957)
- Updates documentation on dynamic item classes. (commit eeb589a)
- Merge pull request #943 from Lazar-T/patch-3 (commit 5fdab02)
- typo (commit b0ae199)
- pywin32 is required by Twisted. closes #937 (commit 5cb0c2f)
- Update install.rst (commit 781286b)
- Merge pull request #928 from Lazar-T/patch-1 (commit b415d04)
- comma instead of fullstop (commit 627b9ba)
- Merge pull request #885 from jsma/patch-1 (commit de90ad)
- Update request-response.rst (commit 3f3263d)
- SgmlLinkExtractor - fix for parsing <area> tag with Unicode present (commit 49b40f0)

Scrappy 0.24.4 (2014-08-09)

- pem file is used by mockserver and required by scrapy bench (commit 5eddc68)
- scrapy bench needs scrapy.tests* (commit d6c999)

Scrappy 0.24.3 (2014-08-09)

- no need to waste travis-ci time on py3 for 0.24 (commit 8e080c1)
- Update installation docs (commit 1d0c096)
- There is a trove classifier for Scrapy framework! (commit 4c701d7)
- update other places where w3lib version is mentioned (commit d109c13)
- Update w3lib requirement to 1.8.0 (commit 39d2ce5)
- Use w3lib.html.replace_entities() (remove_entities() is deprecated) (commit 180d3ad)
• set zip_safe=False (commit a51ee8b)
• do not ship tests package (commit ee3b371)
• scrapy.bat is not needed anymore (commit c3861cf)
• Modernize setup.py (commit 362e322)
• headers can not handle non-string values (commit 94a5c65)
• fix ftp test cases (commit a274a7f)
• The sum up of travis-ci builds are taking like 50min to complete (commit ae1e2cc)
• Update shell.rst typo (commit e49c96a)
• removes weird indentation in the shell results (commit 1ca489d)
• improved explanations, clarified blog post as source, added link for XPath string functions in the spec (commit 65c8f05)
• renamed UserTimeoutError and ServerTimeouterror #583 (commit 037f6ab)
• adding some xpath tips to selectors docs (commit 2d103e0)
• fix tests to account for https://github.com/scrapy/w3lib/pull/23 (commit f8d366a)
• get_func_args maximum recursion fix #728 (commit 81344ea)
• Updated input/output processor example according to #560. (commit f7c4ea8)
• Fixed Python syntax in tutorial. (commit db59ed9)
• Add test case for tunneling proxy (commit f090260)
• Bugfix for leaking Proxy-Authorization header to remote host when using tunneling (commit d8793af)
• Extract links from XHTML documents with MIME-Type “application/xml” (commit ed1f376)
• Merge pull request #793 from roysc/patch-1 (commit 91a1106)
• Fix typo in commands.rst (commit 743e1e2)
• better testcase for settings.overrides.setdefault (commit e22daaf)
• Using CRLF as line marker according to http 1.1 definition (commit 5ec430b)

**Scrapy 0.24.2 (2014-07-08)**

• Use a mutable mapping to proxy deprecated settings.overrides and settings.defaults attribute (commit e5e8133)
• there is not support for python3 yet (commit 3cd6146)
• Update python compatible version set to debian packages (commit fa5d76b)
• DOC fix formatting in release notes (commit c6a9e20)

**Scrapy 0.24.1 (2014-06-27)**

• Fix deprecated CrawlerSettings and increase backwards compatibility with .defaults attribute (commit 8e3f20a)
Scrappy 0.24.0 (2014-06-26)

Enhancements

- Improve Scrappy top-level namespace (issue 494, issue 684)
- Add selector shortcuts to responses (issue 554, issue 690)
- Add new lxml based LinkExtractor to replace unmaintained SgmlLinkExtractor (issue 559, issue 761, issue 763)
- Cleanup settings API - part of per-spider settings GSoC project (issue 737)
- Add UTF8 encoding header to templates (issue 688, issue 762)
- Telnet console now binds to 127.0.0.1 by default (issue 699)
- Update debian/ubuntu install instructions (issue 509, issue 549)
- Disable smart strings in lxml XPath evaluations (issue 535)
- Restore filesystem based cache as default for http cache middleware (issue 541, issue 500, issue 571)
- Expose current crawler in Scrappy shell (issue 557)
- Improve testsuite comparing CSV and XML exporters (issue 570)
- New offsite/filtered and offsite/domains stats (issue 566)
- Support process_links as generator in CrawlSpider (issue 555)
- Verboso logging and new stats counters for DupeFilter (issue 553)
- Add a mimetype parameter to MailSender.send() (issue 602)
- Generalize file pipeline log messages (issue 622)
- Replace unencodeable codepoints with html entities in SGMLLinkExtractor (issue 565)
- Converted SEP documents to rst format (issue 629, issue 630, issue 638, issue 632, issue 636, issue 640, issue 634, issue 639, issue 637, issue 631, issue 641, issue 642)
- Tests and docs for clickdata’s nr index in FormRequest (issue 646, issue 645)
- Allow to disable a downloader handler just like any other component (issue 650)
- Log when a request is discarded after too many redirections (issue 654)
- Log error responses if they are not handled by spider callbacks (issue 612, issue 656)
- Add content-type check to http compression mw (issue 193, issue 660)
- Run pypy tests using latest pypi from ppa (issue 674)
- Run test suite using pytest instead of trial (issue 679)
- Build docs and check for dead links in tox environment (issue 687)
- Make scrapy.version_info a tuple of integers (issue 681, issue 692)
- Infer exporter’s output format from filename extensions (issue 546, issue 659, issue 760)
- Support case-insensitive domains in url_is_from_any_domain() (issue 693)
- Remove pep8 warnings in project and spider templates (issue 698)
- Tests and docs for request_fingerprint function (issue 597)
- Update SEP-19 for GSoC project per-spider settings (issue 705)
- Set exit code to non-zero when contracts fails (issue 727)
• Add a setting to control what class is instantiated as Downloader component (issue 738)
• Pass response in item_dropped signal (issue 724)
• Improve scrapy check contracts command (issue 733, issue 752)
• Document spider.closed() shortcut (issue 719)
• Document request_scheduled signal (issue 746)
• Add a note about reporting security issues (issue 697)
• Add LevelDB http cache storage backend (issue 626, issue 500)
• Sort spider list output of scrapy list command (issue 742)
• Multiple documentation enhancements and fixes (issue 575, issue 587, issue 590, issue 596, issue 610, issue 617, issue 618, issue 627, issue 613, issue 643, issue 654, issue 675, issue 663, issue 711, issue 714)

**Bugfixes**

• Encode unicode URL value when creating Links in RegexLinkExtractor (issue 561)
• Ignore None values in ItemLoader processors (issue 556)
• Fix link text when there is an inner tag in SGMLLinkExtractor and HtmlParserLinkExtractor (issue 485, issue 574)
• Fix wrong checks on subclassing of deprecated classes (issue 581, issue 584)
• Handle errors caused by inspect.stack() failures (issue 582)
• Fix a reference to nonexistent engine attribute (issue 593, issue 594)
• Fix dynamic itemclass example usage of type() (issue 603)
• Use lucasdemarchi/codespell to fix typos (issue 628)
• Fix default value of attrs argument in SgmlLinkExtractor to be tuple (issue 661)
• Fix XXE flaw in sitemap reader (issue 676)
• Fix engine to support filtered start requests (issue 707)
• Fix offsite middleware case on urls with no hostnames (issue 745)
• Testsuite doesn’t require PIL anymore (issue 585)

**Scrapy 0.22.2 (released 2014-02-14)**

• fix a reference to nonexistent engine.slots. closes #593 (commit 13c099a)
• downloaderMW doc typo (spiderMW doc copy remnant) (commit 8ae11bf)
• Correct typos (commit 1346037)

**Scrapy 0.22.1 (released 2014-02-08)**

• localhost666 can resolve under certain circumstances (commit 2ec2279)
• test inspect.stack failure (commit cc3eda3)
• Handle cases when inspect.stack() fails (commit 8cb44f9)
• Fix wrong checks on subclassing of deprecated classes. closes #581 (commit 46d98d6)
• Docs: 4-space indent for final spider example (commit 13846de)
• Fix HtmlParserLinkExtractor and tests after #485 merge (commit 368a946)
• BaseSgmlLinkExtractor: Fixed the missing space when the link has an inner tag (commit b56388)
• BaseSgmlLinkExtractor: Added unit test of a link with an inner tag (commit c1cb418)
• BaseSgmlLinkExtractor: Fixed unknown_endtag() so that it only set current_link=None when the end tag match
the opening tag (commit 7e4d627)
• Fix tests for Travis-CI build (commit 76c7e20)
• replace unencodeable codepoints with html entities. fixes #562 and #285 (commit 5f87b17)
• RegexLinkExtractor: encode URL unicode value when creating Links (commit d0ee545)
• Updated the tutorial crawl output with latest output. (commit 8da65de)
• Updated shell docs with the crawler reference and fixed the actual shell output. (commit 875b9ab)
• PEP8 minor edits. (commit f89efaf)
• Expose current crawler in the scrapy shell. (commit 5349cecf)
• Unused re import and PEP8 minor edits. (commit 387f414)
• Ignore None’s values when using the ItemLoader. (commit 0632546)
• DOC Fixed HTTPCACHE_STORAGE typo in the default value which is now Filesystem instead Dbm. (commit
cde9a8c)
• show ubuntu setup instructions as literal code (commit fb5e9c5)
• Update Ubuntu installation instructions (commit 70fb105)
• Merge pull request #550 from stray-leone/patch-1 (commit 6f70b6a)
• modify the version of scrapy ubuntu package (commit 725900d)
• fix 0.22.0 release date (commit af0219a)
• fix typos in news.rst and remove (not released yet) header (commit b7f58f4)

Scrapy 0.22.0 (released 2014-01-17)

Enhancements

• [Backwards incompatible] Switched HTTPCacheMiddleware backend to filesystem (issue 541) To restore old
backend set HTTPCACHE_STORAGE to scrapy.contrib.httpcache.DbmCacheStorage
• Proxy https:// urls using CONNECT method (issue 392, issue 397)
• Add a middleware to crawl ajax crawleable pages as defined by google (issue 343)
• Rename scrapy.spider.BaseSpider to scrapy.spider.Spider (issue 510, issue 519)
• Selectors register EXSLT namespaces by default (issue 472)
• Unify item loaders similar to selectors renaming (issue 461)
• Make RFPDupeFilter class easily subclassable (issue 533)
• Improve test coverage and forthcoming Python 3 support (issue 525)
• Promote startup info on settings and middleware to INFO level (issue 520)
• Support partials in get_func_args util (issue 506, issue:504)
• Allow running individual tests via tox (issue 503)
• Update extensions ignored by link extractors (issue 498)
• Add middleware methods to get files/images/thumbs paths (issue 490)
• Improve offsite middleware tests (issue 478)
• Add a way to skip default Referer header set by RefererMiddleware (issue 475)
• Do not send x-gzip in default Accept-Encoding header (issue 469)
• Support defining http error handling using settings (issue 466)
• Use modern python idioms wherever you find legacies (issue 497)
• Improve and correct documentation (issue 527, issue 524, issue 521, issue 517, issue 512, issue 505, issue 502, issue 489, issue 465, issue 460, issue 425, issue 536)

Fixes

• Update Selector class imports in CrawlSpider template (issue 484)
• Fix unexistent reference to engine.slots (issue 464)
• Do not try to call body_as_unicode() on a non-TextResponse instance (issue 462)
• Warn when subclassing XPathItemLoader, previously it only warned on instantiation. (issue 523)
• Warn when subclassing XPathSelector, previously it only warned on instantiation. (issue 537)
• Multiple fixes to memory stats (issue 531, issue 530, issue 529)
• Fix overriding url in FormRequest.from_response() (issue 507)
• Fix tests runner under pip 1.5 (issue 513)
• Fix logging error when spider name is unicode (issue 479)

Scrapy 0.20.2 (released 2013-12-09)

• Update CrawlSpider Template with Selector changes (commit 6d1457d)
• fix method name in tutorial. closes GH-480 (commit b4fc359)

Scrapy 0.20.1 (released 2013-11-28)

• include_package_data is required to build wheels from published sources (commit 5ba1ad5)
• process_parallel was leaking the failures on its internal deferreds. closes #458 (commit 419a780)

Scrapy 0.20.0 (released 2013-11-08)

Enhancements

• New Selector’s API including CSS selectors (issue 395 and issue 426),
• Request/Response url/body attributes are now immutable (modifying them had been deprecated for a long time)
• ITEM_PIPELINES is now defined as a dict (instead of a list)
• Sitemap spider can fetch alternate URLs (issue 360)
• Selector.remove_namespaces() now remove namespaces from element’s attributes. (issue 416)
• Paved the road for Python 3.3+ (issue 435, issue 436, issue 431, issue 452)
• New item exporter using native python types with nesting support (issue 366)
• Tune HTTP1.1 pool size so it matches concurrency defined by settings (commit b43b5f575)
• scrapy.mail.MailSender now can connect over TLS or upgrade using STARTTLS (issue 327)
• New FilesPipeline with functionality factored out from ImagesPipeline (issue 370, issue 409)
• Recommend Pillow instead of PIL for image handling (issue 317)
• Added debian packages for Ubuntu quantal and raring (commit 86230c0)
• Mock server (used for tests) can listen for HTTPS requests (issue 410)
• Remove multi spider support from multiple core components (issue 422, issue 421, issue 420, issue 419, issue 423, issue 418)
• Travis-CI now tests Scrapy changes against development versions of w3lib and queueilib python packages.
• Add pypy 2.1 to continuous integration tests (commit ecf7431)
• Pylinted, pep8 and removed old-style exceptions from source (issue 430, issue 432)
• Use importlib for parametric imports (issue 445)
• Handle a regression introduced in Python 2.7.5 that affects XmlItemExporter (issue 372)
• Bugfix crawling shutdown on SIGINT (issue 450)
• Do not submit reset type inputs in FormRequest.from_response (commit b326b87)
• Do not silence download errors when request errback raises an exception (commit 684cfc0)

Bugfixes

• Fix tests under Django 1.6 (commit b6bed44c)
• Lot of bugfixes to retry middleware under disconnections using HTTP 1.1 download handler
• Fix inconsistencies among Twisted releases (issue 406)
• Fix scrapy shell bugs (issue 418, issue 407)
• Fix invalid variable name in setup.py (issue 429)
• Fix tutorial references (issue 387)
• Improve request-response docs (issue 391)
• Improve best practices docs (issue 399, issue 400, issue 401, issue 402)
• Improve django integration docs (issue 404)
• Document bindaddress request meta (commit 37c24e01d7)
• Improve Request class documentation (issue 226)
Other

- Dropped Python 2.6 support (issue 448)
- Add cssselect python package as install dependency
- Drop libxml2 and multi selector’s backend support, lxml is required from now on.
- Minimum Twisted version increased to 10.0.0, dropped Twisted 8.0 support.
- Running test suite now requires mock python library (issue 390)

Thanks

Thanks to everyone who contribute to this release!

List of contributors sorted by number of commits:

<table>
<thead>
<tr>
<th></th>
<th>Name</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>69</td>
<td>Daniel Graña</td>
<td>dangra@...</td>
</tr>
<tr>
<td>37</td>
<td>Pablo Hoffman</td>
<td>pablo@...</td>
</tr>
<tr>
<td>13</td>
<td>Mikhail Korobov</td>
<td>kmike84@...</td>
</tr>
<tr>
<td>9</td>
<td>Alex Cepoi</td>
<td>alex.cepoi@...</td>
</tr>
<tr>
<td>9</td>
<td>alexanderlukanin13</td>
<td>alexander.lukanin.13@...</td>
</tr>
<tr>
<td>8</td>
<td>Rolando Espinoza La fuente</td>
<td>darkrho@...</td>
</tr>
<tr>
<td>8</td>
<td>Lukasz Biedrycki</td>
<td>lukasz.biedrycki@...</td>
</tr>
<tr>
<td>6</td>
<td>Nicolas Ramirez</td>
<td>nramirez.uy@...</td>
</tr>
<tr>
<td>3</td>
<td>Paul Tremberth</td>
<td>paul.tremberth@...</td>
</tr>
<tr>
<td>2</td>
<td>Martin Olveyra</td>
<td>molveyra@...</td>
</tr>
<tr>
<td>2</td>
<td>Stefan</td>
<td>misc@...</td>
</tr>
<tr>
<td>2</td>
<td>Rolando Espinoza La fuente</td>
<td>darkrho@...</td>
</tr>
<tr>
<td>2</td>
<td>Loren Davie</td>
<td>loren@...</td>
</tr>
<tr>
<td>2</td>
<td>irgmedeiros</td>
<td>irgmedeiros@...</td>
</tr>
<tr>
<td>1</td>
<td>Stefan Koch</td>
<td>taikano@...</td>
</tr>
<tr>
<td>1</td>
<td>Stefan</td>
<td>cct@...</td>
</tr>
<tr>
<td>1</td>
<td>scraperdragon</td>
<td>dragon@...</td>
</tr>
<tr>
<td>1</td>
<td>Kumara Tharmalingam</td>
<td>ktharmal@...</td>
</tr>
<tr>
<td>1</td>
<td>Francesco Piccinno</td>
<td>stack.box@...</td>
</tr>
<tr>
<td>1</td>
<td>Marcos Campal</td>
<td>duendex@...</td>
</tr>
<tr>
<td>1</td>
<td>Dragon Dave</td>
<td>dragon@...</td>
</tr>
<tr>
<td>1</td>
<td>Capi Etheriel</td>
<td>barraponto@...</td>
</tr>
<tr>
<td>1</td>
<td>cacovsky</td>
<td>amarquesferraz@...</td>
</tr>
<tr>
<td>1</td>
<td>Berend Iwema</td>
<td>berend@...</td>
</tr>
</tbody>
</table>

Scrapy 0.18.4 (released 2013-10-10)

- IPython refuses to update the namespace. fix #396 (commit 3d32c4f)
- Fix AlreadyCalledError replacing a request in shell command. closes #407 (commit b1d8919)
- Fix start_requests laziness and early hangs (commit 89faf52)

Scrapy 0.18.3 (released 2013-10-03)

- fix regression on lazy evaluation of start requests (commit 12693a5)
- forms: do not submit reset inputs (commit e429f63)
- increase unittest timeouts to decrease travis false positive failures (commit 912202e)
Scrapy Documentation, Release 1.2.3

- backport master fixes to json exporter (commit cfc2d46)
- Fix permission and set umask before generating sdist tarball (commit 06149e0)

Scrapy 0.18.2 (released 2013-09-03)

- Backport scrapy check command fixes and backward compatible multi crawler process (issue 339)

Scrapy 0.18.1 (released 2013-08-27)

- remove extra import added by cherry picked changes (commit d20304e)
- fix crawling tests under twisted pre 11.0.0 (commit 1994f38)
- py26 can not format zero length fields {} (commit abf756f)
- test PotentialDataLoss errors on unbound responses (commit b15470d)
- Treat responses without content-length or Transfer-Encoding as good responses (commit c4bf324)
- do no include ResponseFailed if http1 handler is not enabled (commit 6cbe684)
- New HTTP client wraps connection losts in ResponseFailed exception. fix #373 (commit ia20bb)
- limit travis-ci build matrix (commit 3b01bb8)
- Merge pull request #375 from peterarennot/patch-1 (commit fa766d7)
- Fixed so it refers to the correct folder (commit 3283809)
- added quantal & raring to support ubuntu releases (commit 1411923)
- fix retry middleware which didn’t retry certain connection errors after the upgrade to http1 client, closes GH-373 (commit bb35e0)
- fix XmlItemExporter in Python 2.7.4 and 2.7.5 (commit de3e451)
- minor updates to 0.18 release notes (commit c45e5f1)
- fix contributers list format (commit 0b60031)

Scrapy 0.18.0 (released 2013-08-09)

- Lot of improvements to testsuite run using Tox, including a way to test on pypi
- Handle GET parameters for AJAX crawleable urls (commit 3fe2a32)
- Use xml recover option to parse sitemaps (issue 347)
- Bugfix cookie merging by hostname and not by netloc (issue 352)
- Support disabling HttpCompressionMiddleware using a flag setting (issue 359)
- Support xml namespaces using iernodes parser in XMLFeedSpider (issue 12)
- Support dont_cache request meta flag (issue 19)
- Bugfix scrapy.utils.gz.gunzip broken by changes in python 2.7.4 (commit 4dc76e)
- Bugfix url encoding on SgmlLinkExtractor (issue 24)
- Bugfix TakeFirst processor shouldn’t discard zero (0) value (issue 59)
- Support nested items in xml exporter (issue 66)

7.1. Release notes
• Improve cookies handling performance (issue 77)
• Log dupe filtered requests once (issue 105)
• Split redirection middleware into status and meta based middlewares (issue 78)
• Use HTTP1.1 as default downloader handler (issue 109 and issue 318)
• Support xpath form selection on FormRequest.from_response (issue 185)
• Bugfix unicode decoding error on SgmlLinkExtractor (issue 199)
• Bugfix signal dispatching on pypi interpreter (issue 205)
• Improve request delay and concurrency handling (issue 206)
• Add RFC2616 cache policy to HttpCacheMiddleware (issue 212)
• Allow customization of messages logged by engine (issue 214)
• Multiples improvements to DjangoItem (issue 217, issue 218, issue 221)
• Extend Scrapy commands using setuptools entry points (issue 260)
• Allow spider allowed_domains value to be set/tuple (issue 261)
• Support settings.getdict (issue 269)
• Simplify internal scrapy.core.scrapers slot handling (issue 271)
• Added Item.copy (issue 290)
• Collect idle downloader slots (issue 297)
• Add ftp:// scheme downloader handler (issue 329)
• Added downloader benchmark webserver and spider tools Benchmarking
• Moved persistent (on disk) queues to a separate project (queuelib) which scrapy now depends on
• Add scrapy commands using external libraries (issue 260)
• Added --pdb option to scrapy command line tool
• Added XPathSelector.remove_namespaces() which allows to remove all namespaces from XML documents for convenience (to work with namespace-less XPaths). Documented in Selectors.
• Several improvements to spider contracts
• New default middleware named MetaRefreshMiddleware that handles meta-refresh html tag redirections,
• MetaRefreshMiddleware and RedirectMiddleware have different priorities to address #62
• added from_crawler method to spiders
• added system tests with mock server
• more improvements to Mac OS compatibility (thanks Alex Cepoi)
• several more cleanups to singletons and multi-spider support (thanks Nicolas Ramirez)
• support custom download slots
• added --spider option to “shell” command.
• log overridden settings when scrapy starts

Thanks to everyone who contribute to this release. Here is a list of contributors sorted by number of commits:
Scrapy 0.16.5 (released 2013-05-30)

- obey request method when scrapy deploy is redirected to a new endpoint (commit 8c4fcee)
- fix inaccurate downloader middleware documentation. refs #280 (commit 40667cb)
- doc: remove links to diveintopython.org, which is no longer available. closes #246 (commit bd58bfa)
- Find form nodes in invalid html5 documents (commit e3d6945)
- Fix typo labeling attrs type bool instead of list (commit a274276)

Scrapy 0.16.4 (released 2013-01-23)

- fixes spelling errors in documentation (commit 6d2b3aa)
- add doc about disabling an extension. refs #132 (commit e90de33)
Scrapy Documentation, Release 1.2.3

- Fixed error message formatting. log.err() doesn’t support cool formatting and when error occurred, the message was: “ERROR: Error processing %(item)s” (commit c16150c)
- lint and improve images pipeline error logging (commit 56b45fc)
- fixed doc typos (commit 243be84)
- add documentation topics: Broad Crawls & Common Practies (commit 1fbb715)
- fix bug in scrapy parse command when spider is not specified explicitly. closes #209 (commit c72e682)
- Update docs/topics/commands.rst (commit 28eac7a)

Scrapy 0.16.3 (released 2012-12-07)

- Remove concurrency limitation when using download delays and still ensure inter-request delays are enforced (commit 487b9b5)
- add error details when image pipeline fails (commit 8232569)
- improve mac os compatibility (commit 8dcf8aa)
- setup.py: use README.rst to populate long_description (commit 7b5310d)
- doc: removed obsolete references to ClientForm (commit 80f9bb6)
- correct docs for default storage backend (commit 2aa491b)
- doc: removed broken proxyhub link from FAQ (commit bdf61c4)
- Fixed docs typo in SpiderOpenCloseLogging example (commit 7184094)

Scrapy 0.16.2 (released 2012-11-09)

- scrapy contracts: python2.6 compat (commit a4a9199)
- scrapy contracts verbose option (commit ec41673)
- proper unittest-like output for scrapy contracts (commit 86635e4)
- added open_in_browser to debugging doc (commit c9b690d)
- removed reference to global scrapy stats from settings doc (commit dd55067)
- Fix SpiderState bug in Windows platforms (commit 58998f4)

Scrapy 0.16.1 (released 2012-10-26)

- fixed LogStats extension, which got broken after a wrong merge before the 0.16 release (commit 8c780fd)
- better backwards compatibility for scrapy.conf.settings (commit 3403089)
- extended documentation on how to access crawler stats from extensions (commit c4da0b5)
- removed .hgtags (no longer needed now that scrapy uses git) (commit d52c188)
- fix dashes under rst headers (commit fa4f7f9)
- set release date for 0.16.0 in news (commit e292246)
Scrapy 0.16.0 (released 2012-10-18)

Scrapy changes:

- added *Spiders Contracts*, a mechanism for testing spiders in a formal/reproducible way
- added options \(-o\) and \(-t\) to the *runspider* command
- documented *AutoThrottle extension* and added to extensions installed by default. You still need to enable it with `AUTOThROTTLE_ENABLED`
- major Stats Collection refactoring: removed separation of global/per-spider stats, removed stats-related signals (`stats_spider_opened`, etc). Stats are much simpler now, backwards compatibility is kept on the Stats Collector API and signals.
- added `process_start_requests()` method to spider middlewares
- dropped Signals singleton. Signals should now be accessed through the Crawler.signals attribute. See the signals documentation for more info.
- dropped Signals singleton. Signals should now be accessed through the Crawler.signals attribute. See the signals documentation for more info.
- dropped Stats Collector singleton. Stats can now be accessed through the Crawler.stats attribute. See the stats collection documentation for more info.
- documented *Core API*
- *lxml* is now the default selectors backend instead of *libxml2*
- ported FormRequest.from_response() to use *lxml* instead of *ClientForm*
- removed modules: `scrapy.xlib.BeautifulSoup` and `scrapy.xlib.ClientForm`
- SitemapSpider: added support for sitemap urls ending in `.xml` and `.xml.gz`, even if they advertise a wrong content type (commit 10ed28b)
- StackTraceDump extension: also dump trackref live references (commit fe2ce93)
- nested items now fully supported in JSON and JSONLines exporters
- added `cookiejar` Request meta key to support multiple cookie sessions per spider
- decoupled encoding detection code to *w3lib.encoding*, and ported Scrapy code to use that module
- dropped support for Python 2.5. See https://blog.scrapinghub.com/2012/02/27/scrapy-0-15-dropping-support-for-python-2-5/
- dropped support for Twisted 2.5
- added `REFERER_ENABLED` setting, to control referer middleware
- changed default user agent to: `Scrapy/VERSION (+http://scrapy.org)`
- removed (undocumented) `HTMLImageLinkExtractor` class from `scrapy.contrib.linkextractors.image`
- removed per-spider settings (to be replaced by instantiating multiple crawler objects)
- `USER_AGENT` spider attribute will no longer work, use `user_agent` attribute instead
- `DOWNLOAD_TIMEOUT` spider attribute will no longer work, use `download_timeout` attribute instead
- removed `ENCODING_ALIASES` setting, as encoding auto-detection has been moved to the `w3lib` library
- promoted topics-djangoitem to main contrib
• LogFormatter method now return dicts (instead of strings) to support lazy formatting (issue 164, commit dcef7b0)
• downloader handlers (DOWNLOAD_HANDLERS setting) now receive settings as the first argument of the constructor
• replaced memory usage accounting with (more portable) resource module, removed scrapy.utils.memory module
• removed signal: scrapy.mail.mail_sent
• removed TRACK_REFS setting, now trackrefs is always enabled
• DBM is now the default storage backend for HTTP cache middleware
• number of log messages (per level) are now tracked through Scrapy stats (stat name: log_count/LEVEL)
• number received responses are now tracked through Scrapy stats (stat name: response_received_count)
• removed scrapy.log.started attribute

Scrapy 0.14.4

• added precise to supported ubuntu distros (commit b7e46df)
• fixed bug in json-rpc webservice reported in https://groups.google.com/forum/#!topic/scrapy-users/qgVBmFybNAQ/discussion. also removed no longer supported ‘run’ command from extras/scrapy-ws.py (commit 340fbdh)
• meta tag attributes for content-type http equiv can be in any order. #123 (commit 0cb68af)
• replace “import Image” by more standard “from PIL import Image”. closes #88 (commit 4d17048)
• return trial status as bin/runtests.sh exit value. #118 (commit b7b2e7f)

Scrapy 0.14.3

• forgot to include pydispatch license. #118 (commit fd85f9c)
• include egg files used by testsuite in source distribution. #118 (commit c897793)
• update docstring in project template to avoid confusion with genspider command, which may be considered as an advanced feature. refs #107 (commit 2548dce)
• added note to docs/topics/firebug.rst about google directory being shut down (commit 668e352)
• dont discard slot when empty, just save in another dict in order to recycle if needed again. (commit 8e9f607)
• do not fail handling unicode xpaths in libxml2 backed selectors (commit b830e95)
• fixed minor mistake in Request objects documentation (commit bf3c9ee)
• fixed minor defect in link extractors documentation (commit ba14f38)
• removed some obsolete remaining code related to sqlite support in scrapy (commit 0665175)

Scrapy 0.14.2

• move buffer pointing to start of file before computing checksum. refs #92 (commit 6a5bef2)
• Compute image checksum before persisting images. closes #92 (commit 9817d1f)
• remove leaking references in cached failures (commit 673a120)
• fixed bug in MemoryUsage extension: get_engine_status() takes exactly 1 argument (0 given) (commit 11133e9)
• fixed struct.error on http compression middleware. closes #87 (commit 1423140)
• ajax crawling wasn’t expanding for unicode urls (commit 0dc3fb4)
• Catch start_requests iterator errors. refs #83 (commit 454a21d)
• Speed-up libxml2 XPathSelector (commit 2fbd662)
• updated versioning doc according to recent changes (commit 0a070f5)
• scrapyd: fixed documentation link (commit 2b4e4c3)
• extras/makedeb.py: no longer obtaining version from git (commit caffe0e)

Scrapy 0.14.1

• extras/makedeb.py: no longer obtaining version from git (commit caffe0e)
• bumped version to 0.14.1 (commit 6cb9e1c)
• fixed reference to tutorial directory (commit 4b86bd6)
• doc: removed duplicated callback argument from Request.replace() (commit 1aecddd)
• fixed formatting of scrapyd doc (commit 8bf19e6)
• Dump stacks for all running threads and fix engine status dumped by StackTraceDump extension (commit 14a8e6e)
• added comment about why we disable ssl on boto images upload (commit 5223575)
• SSL handshaking hangs when doing too many parallel connections to S3 (commit 63d583d)
• change tutorial to follow changes on dmoz site (commit bcb3198)
• Avoid _disconnectedDeferred AttributeError exception in Twisted>=11.1.0 (commit 98f3f87)
• allow spider to set autothrottle max concurrency (commit 175a4b5)

Scrapy 0.14

New features and settings

• Support for AJAX crawlable urls
• New persistent scheduler that stores requests on disk, allowing to suspend and resume crawls (r2737)
• added -o option to scrapy crawl, a shortcut for dumping scraped items into a file (or standard output using -)
• Added support for passing custom settings to Scrapyd schedule.json api (r2779, r2783)
• New ChunkedTransferMiddleware (enabled by default) to support chunked transfer encoding (r2769)
• Add boto 2.0 support for S3 downloader handler (r2763)
• Added marshal to formats supported by feed exports (r2744)
• In request errbacks, offending requests are now received in failure.request attribute (r2738)
• Big downloader refactoring to support per domain/ip concurrency limits (r2732)
• Added builtin caching DNS resolver (r2728)
• Moved Amazon AWS-related components/extensions (SQS spider queue, SimpleDB stats collector) to a separate project: [scaws](https://github.com/scrapinghub/scaws) (r2706, r2714)
• Moved spider queues to scrapyd: `scrapy.spiderqueue -> scrapyd.spiderqueue` (r2708)
• Moved sqlite utils to scrapyd: `scrapy.utils.sqlite -> scrapyd.sqlite` (r2781)
• Real support for returning iterators on `start_requests()` method. The iterator is now consumed during the crawl when the spider is getting idle (r2704)
• Added `REDIRECT_ENABLED` setting to quickly enable/disable the redirect middleware (r2697)
• Added `RETRY_ENABLED` setting to quickly enable/disable the retry middleware (r2694)
• Added `CloseSpider` exception to manually close spiders (r2691)
• Improved encoding detection by adding support for HTML5 meta charset declaration (r2690)
• Refactored close spider behavior to wait for all downloads to finish and be processed by spiders, before closing the spider (r2688)
• Added `SitemapSpider` (see documentation in Spiders page) (r2658)
• Added `LogStats` extension for periodically logging basic stats (like crawled pages and scraped items) (r2657)
• Make handling of gzipped responses more robust (#319, r2643). Now Scrapy will try and decompress as much as possible from a gzipped response, instead of failing with an IOError.
• Simplified `MemoryDebugger` extension to use stats for dumping memory debugging info (r2639)
• Added new command to edit spiders: `scrapy edit` (r2636) and `-e` flag to `genspider` command that uses it (r2653)
• Changed default representation of items to pretty-printed dicts. (r2631). This improves default logging by making log more readable in the default case, for both Scraped and Dropped lines.
• Added `spider_error` signal (r2628)
• Added `COOKIES_ENABLED` setting (r2625)
• Stats are now dumped to Scrapy log (default value of `STATS_DUMP` setting has been changed to `True`). This is to make Scrapy users more aware of Scrapy stats and the data that is collected there.
• Added support for dynamically adjusting download delay and maximum concurrent requests (r2599)
• Added new DBM HTTP cache storage backend (r2576)
• Added `listjobs.json` API to Scrapyd (r2571)
• CsvItemExporter: added `join_multivalued` parameter (r2578)
• Added namespace support to `xmliter_lxml` (r2552)
• Improved cookies middleware by making `COOKIES_DEBUG` nicer and documenting it (r2579)
• Several improvements to Scrapyd and Link extractors
Code rearranged and removed

- Merged item passed and item scraped concepts, as they have often proved confusing in the past. This means: (r2630)
  - original item_scraped signal was removed
  - original item_passed signal was renamed to item_scraped
  - old log lines Scraped Item... were removed
  - old log lines Passed Item... were renamed to Scraped Item... lines and downgraded to DEBUG level

- Reduced Scrapy codebase by striping part of Scrapy code into two new libraries:
  - w3lib (several functions from scrapy.utils.{http,markup,multipart,response, url}, done in r2584)
  - scrapely (was scrapy.contrib.ibl, done in r2586)

- Removed unused function: scrapy.utils.request.request_info() (r2577)

- Removed googledir project from examples/googledir. There’s now a new example project called dirbot available on github: https://github.com/scrapy/dirbot

- Removed support for default field values in Scrapy items (r2616)

- Removed experimental crawlspider v2 (r2632)

- Removed scheduler middleware to simplify architecture. Duplicates filter is now done in the scheduler itself, using the same dupe filtering class as before (DUPEFILTER_CLASS setting) (r2640)

- Removed support for passing urls to scrapy crawl command (use scrapy parse instead) (r2704)

- Removed deprecated Execution Queue (r2704)

- Removed (undocumented) spider context extension (from scrapy.contrib.spidercontext) (r2780)

- removed CONCURRENT_SPIDERS setting (use scrapyd maxproc instead) (r2789)

- Renamed attributes of core components: downloader.sites -> downloader.slots, scraper.sites -> scraper.slots (r2717, r2718)

- Renamed setting CLOSESPIDER_ITEMPASSED to CLOSESPIDER_ITEMCOUNT (r2655). Backwards compatibility kept.

Scrapy 0.12

The numbers like #NNN reference tickets in the old issue tracker (Trac) which is no longer available.

New features and improvements

- Passed item is now sent in the item argument of the item_passed (#273)
- Added verbose option to scrapy version command, useful for bug reports (#298)
- HTTP cache now stored by default in the project data dir (#279)
- Added project data storage directory (#276, #277)
- Documented file structure of Scrapy projects (see command-line tool doc)
- New lxml backend for XPath selectors (#147)
• Per-spider settings (#245)
• Support exit codes to signal errors in Scrapy commands (#248)
• Added -c argument to scrapy shell command
• Made libxml2 optional (#260)
• New deploy command (#261)
• Added CLOSESPIDER_PAGECOUNT setting (#253)
• Added CLOSESPIDER_ERRORCOUNT setting (#254)

Scrapy changes

• Scrapy now uses one process per spider
• It stores one log file per spider run, and rotate them keeping the lastest 5 logs per spider (by default)
• A minimal web ui was added, available at http://localhost:6800 by default
• There is now a scrapy server command to start a Scrapyd server of the current project

Changes to settings

• added HTTPCACHE_ENABLED setting (False by default) to enable HTTP cache middleware
• changed HTTPCACHE_EXPIRATION_SECS semantics: now zero means “never expire”.

Deprecated/obsoleted functionality

• Deprecated runserver command in favor of server command which starts a Scrapy server. See also: Scrapyd changes
• Deprecated queue command in favor of using Scrapyd schedule.json API. See also: Scrapyd changes
• Removed the !LxmlItemLoader (experimental contrib which never graduated to main contrib)

Scrapy 0.10

The numbers like #NNN reference tickets in the old issue tracker (Trac) which is no longer available.

New features and improvements

• New Scrapy service called scrapyd for deploying Scrapy crawlers in production (#218) (documentation available)
• Simplified Images pipeline usage which doesn’t require subclassing your own images pipeline now (#217)
• Scrapy shell now shows the Scrapy log by default (#206)
• Refactored execution queue in a common base code and pluggable backends called “spider queues” (#220)
• New persistent spider queue (based on SQLite) (#198), available by default, which allows to start Scrapy in server mode and then schedule spiders to run.
• Added documentation for Scrapy command-line tool and all its available sub-commands. (documentation available)
• Feed exporters with pluggable backends (#197) (documentation available)
• Deferred signals (#193)
• Added two new methods to item pipeline open_spider(), close_spider() with deferred support (#195)
• Support for overriding default request headers per spider (#181)
• Replaced default Spider Manager with one with similar functionality but not depending on Twisted Plugins (#186)
• Splitted Debian package into two packages - the library and the service (#187)
• Scrapy log refactoring (#188)
• New extension for keeping persistent spider contexts among different runs (#203)
• Added `dont_redirect` request.meta key for avoiding redirects (#233)
• Added `dont_retry` request.meta key for avoiding retries (#234)

**Command-line tool changes**

• New `scrapy` command which replaces the old `scrapy-ctl.py` (#199) - there is only one global `scrapy` command now, instead of one `scrapy-ctl.py` per project - Added `scrapy.bat` script for running more conveniently from Windows
• Added bash completion to command-line tool (#210)
• Renamed command `start` to `runserver` (#209)

**API changes**

• `url` and `body` attributes of Request objects are now read-only (#230)
• `Request.copy()` and `Request.replace()` now also copies their `callback` and `errback` attributes (#231)
• Removed `UrlFilterMiddleware` from `scrapy.contrib` (already disabled by default)
• Offsite middleware doesn’t filter out any request coming from a spider that doesn’t have a allowed_domains attribute (#225)
• Removed Spider Manager `load()` method. Now spiders are loaded in the constructor itself.
• Changes to Scrapy Manager (now called “Crawler”):
  - `scrapy.core.manager.ScrapyManager` class renamed to `scrapy.crawler.Crawler`
  - `scrapy.core.manager.scrapymanager` singleton moved to `scrapy.project.crawler`
• Moved module: `scrapy.contrib.spidermanager` to `scrapy.spidermanager`
• Spider Manager singleton moved from `scrapy.spider.spiders` to the spiders’ attribute of `scrapy.project.crawler` singleton.
• moved Stats Collector classes: (#204)
  - `scrapy.stats.collector.StatsCollector` to `scrapy.statscol.StatsCollector`
  - `scrapy.stats.collector.SimpledbStatsCollector` to `scrapy.contrib.statscol.SimpledbStatsCollector`
• default per-command settings are now specified in the `default_settings` attribute of command object class (#201)

• changed arguments of Item pipeline `process_item()` method from `(spider, item)` to `(item, spider)`
  
  – backwards compatibility kept (with deprecation warning)

• moved `scrapy.core.signals` module to `scrapy.signals`
  
  – backwards compatibility kept (with deprecation warning)

• moved `scrapy.core.exceptions` module to `scrapy.exceptions`
  
  – backwards compatibility kept (with deprecation warning)

• added `handles_request()` class method to `BaseSpider`

• dropped `scrapy.log.exc()` function (use `scrapy.log.err()` instead)

• dropped component argument of `scrapy.log.msg()` function

• dropped `scrapy.log.log_level` attribute

• Added `from_settings()` class methods to Spider Manager, and Item Pipeline Manager

Changes to settings

• Added `HTTPCACHE_IGNORE_SCHEMES` setting to ignore certain schemes on !HttpCacheMiddleware (#225)

• Added `SPIDER_QUEUE_CLASS` setting which defines the spider queue to use (#220)

• Added `KEEP_ALIVE` setting (#220)

• Removed `SERVICE_QUEUE` setting (#220)

• Removed `COMMANDS_SETTINGS_MODULE` setting (#201)

• Renamed `REQUEST_HANDLERS` to `DOWNLOAD_HANDLERS` and make download handlers classes (instead of functions)

Scrapy 0.9

The numbers like #NNN reference tickets in the old issue tracker (Trac) which is no longer available.

New features and improvements

• Added SMTP-AUTH support to `scrapy.mail`

• New settings added: `MAIL_USER, MAIL_PASS` (r2065 | #149)

• Added new scrapy-ctl view command - To view URL in the browser, as seen by Scrapy (r2039)

• Added web service for controlling Scrapy process (this also deprecates the web console. (r2053 | #167)

• Support for running Scrapy as a service, for production systems (r1988, r2054, r2055, r2056, r2057 | #168)

• Added wrapper induction library (documentation only available in source code for now). (r2011)

• Simplified and improved response encoding support (r1961, r1969)

• Added `LOG_ENCODING` setting (r1956, documentation available)

• Added `RANDOMIZE_DOWNLOAD_DELAY` setting (enabled by default) (r1923, doc available)
• **MailSender** is no longer IO-blocking (r1955 | #146)
• Linkextractors and new Crawlsper now handle relative base tag urls (r1960 | #148)
• Several improvements to Item Loaders and processors (r2022, r2023, r2024, r2025, r2026, r2027, r2028, r2029, r2030)
• Added support for adding variables to telnet console (r2047 | #165)
• Support for requests without callbacks (r2050 | #166)

**API changes**

• Change Spider.domain_name to Spider.name (SEP-012, r1975)
• Response.encoding is now the detected encoding (r1961)
• **HttpErrorMiddleware** now returns None or raises an exception (r2006 | #157)
• scrapy.command modules relocation (r2035, r2036, r2037)
• Added **ExecutionQueue** for feeding spiders to scrape (r2034)
• Removed **ExecutionEngine** singleton (r2039)
• Ported **S3ImagesStore** (images pipeline) to use boto and threads (r2033)
• Moved module: scrapy.management.telnet to scrapy.telnet (r2047)

**Changes to default settings**

• Changed default **SCHEDULER_ORDER** to DFO (r1939)

**Scrapy 0.8**

The numbers like #NNN reference tickets in the old issue tracker (Trac) which is no longer available.

**New features**

• Added **DEFAULT_RESPONSE_ENCODING** setting (r1809)
• Added **dont_click** argument to FormRequest.from_response() method (r1813, r1816)
• Added **clickdata** argument to FormRequest.from_response() method (r1802, r1803)
• Added support for HTTP proxies (**HttpProxyMiddleware**) (r1781, r1785)
• Offsite spider middleware now logs messages when filtering out requests (r1841)

**Backwards-incompatible changes**

• Changed scrapy.utils.response.get_meta_refresh() signature (r1804)
• Removed deprecated scrapy.item.ScrapedItem class - use scrapy.item.Item instead (r1838)
• Removed deprecated scrapy.xpath module - use scrapy.selector instead. (r1836)
• Removed deprecated core.signals.domain_open signal - use core.signals.domain_opened instead (r1822)
• **log.msg()** now receives a **spider** argument *(r1822)*
  – Old domain argument has been deprecated and will be removed in 0.9. For spiders, you should always use the **spider** argument and pass spider references. If you really want to pass a string, use the component argument instead.

• Changed core signals **domain_opened**, **domain_closed**, **domain_idle**

• Changed Item pipeline to use spiders instead of domains
  – The domain argument of **process_item()** item pipeline method was changed to **spider**, the new signature is: **process_item(spider, item)** *(r1827 | #105)*
  – To quickly port your code (to work with Scrapy 0.8) just use **spider.domain_name** where you previously used **domain**.

• Changed Stats API to use spiders instead of domains *(r1849 | #113)*
  – **StatsCollector** was changed to receive spider references (instead of domains) in its methods (**set_value**, **inc_value**, etc).
  – added **StatsCollector.iter_spider_stats()** method
  – removed **StatsCollector.list_domains()** method
  – Also, Stats signals were renamed and now pass around spider references (instead of domains). Here’s a summary of the changes:
    – To quickly port your code (to work with Scrapy 0.8) just use **spider.domain_name** where you previously used **domain**. **spider_stats** contains exactly the same data as **domain_stats**.

• **CloseDomain** extension moved to **scrapy.contrib.closespider.CloseSpider** *(r1833)*
  – Its settings were also renamed:
    * **CLOSEDOMAIN_TIMEOUT** to **CLOSESPIDER_TIMEOUT**
    * **CLOSEDOMAIN_ITEMCOUNT** to **CLOSESPIDER_ITEMCOUNT**

• Removed deprecated **SCRAPYSETTINGS_MODULE** environment variable - use **SCRAPI_SETTINGS_MODULE** instead *(r1840)*

• Renamed setting: **REQUESTS_PER_DOMAIN** to **CONCURRENT_REQUESTS_PER_SPIDER** *(r1830, r1844)*

• Renamed setting: **CONCURRENT_DOMAINS** to **CONCURRENT_SPIDERS** *(r1830)*

• Refactored HTTP Cache middleware
  – HTTP Cache middleware has been heavily refactored, retaining the same functionality except for the domain sectorization which was removed. *(r1843)*

• Renamed exception: **DontCloseDomain** to **DontCloseSpider** *(r1859 | #120)*

• Renamed extension: **DelayedCloseDomain** to **SpiderCloseDelay** *(r1861 | #121)*

• Removed obsolete **scrapy.utils.markup.remove_escape_chars** function - use **scrapy.utils.markup.replace_escape_chars** instead *(r1865)*

### Scrapy 0.7

First release of Scrapy.
Contributing to Scrapy

Important: Double check you are reading the most recent version of this document at http://doc.scrapy.org/en/master/contributing.html

There are many ways to contribute to Scrapy. Here are some of them:

- Blog about Scrapy. Tell the world how you’re using Scrapy. This will help newcomers with more examples and the Scrapy project to increase its visibility.
- Report bugs and request features in the issue tracker, trying to follow the guidelines detailed in Reporting bugs below.
- Submit patches for new functionality and/or bug fixes. Please read Writing patches and Submitting patches below for details on how to write and submit a patch.
- Join the scrapy-users mailing list and share your ideas on how to improve Scrapy. We’re always open to suggestions.

Reporting bugs

Note: Please report security issues only to scrapy-security@googlegroups.com. This is a private list only open to trusted Scrapy developers, and its archives are not public.

Well-written bug reports are very helpful, so keep in mind the following guidelines when reporting a new bug.

- check the FAQ first to see if your issue is addressed in a well-known question
- check the open issues to see if it has already been reported. If it has, don’t dismiss the report but check the ticket history and comments, you may find additional useful information to contribute.
- search the scrapy-users list to see if it has been discussed there, or if you’re not sure if what you’re seeing is a bug. You can also ask in the #scrapy IRC channel.
- write complete, reproducible, specific bug reports. The smaller the test case, the better. Remember that other developers won’t have your project to reproduce the bug, so please include all relevant files required to reproduce it. See for example StackOverflow’s guide on creating a Minimal, Complete, and Verifiable example exhibiting the issue.
- include the output of scrapy version -v so developers working on your bug know exactly which version and platform it occurred on, which is often very helpful for reproducing it, or knowing if it was already fixed.

Writing patches

The better written a patch is, the higher chance that it’ll get accepted and the sooner that will be merged.

Well-written patches should:

- contain the minimum amount of code required for the specific change. Small patches are easier to review and merge. So, if you’re doing more than one change (or bug fix), please consider submitting one patch per change. Do not collapse multiple changes into a single patch. For big changes consider using a patch queue.
- pass all unit-tests. See Running tests below.
- include one (or more) test cases that check the bug fixed or the new functionality added. See Writing tests below.
• if you’re adding or changing a public (documented) API, please include the documentation changes in the same patch. See Documentation policies below.

Submitting patches

The best way to submit a patch is to issue a pull request on GitHub, optionally creating a new issue first.

Remember to explain what was fixed or the new functionality (what it is, why it’s needed, etc). The more info you include, the easier will be for core developers to understand and accept your patch.

You can also discuss the new functionality (or bug fix) before creating the patch, but it’s always good to have a patch ready to illustrate your arguments and show that you have put some additional thought into the subject. A good starting point is to send a pull request on GitHub. It can be simple enough to illustrate your idea, and leave documentation/tests for later, after the idea has been validated and proven useful. Alternatively, you can send an email to scrapy-users to discuss your idea first. When writing GitHub pull requests, try to keep titles short but descriptive. E.g. For bug #411: “Scrapy hangs if an exception raises in start_requests” prefer “Fix hanging when exception occurs in start_requests (#411)” instead of “Fix for #411”. Complete titles make it easy to skim through the issue tracker.

Finally, try to keep aesthetic changes (PEP 8 compliance, unused imports removal, etc) in separate commits than functional changes. This will make pull requests easier to review and more likely to get merged.

Coding style

Please follow these coding conventions when writing code for inclusion in Scrapy:

• Unless otherwise specified, follow PEP 8.

• It’s OK to use lines longer than 80 chars if it improves the code readability.

• Don’t put your name in the code you contribute. Our policy is to keep the contributor’s name in the AUTHORS file distributed with Scrapy.

Scrapy Contrib

Scrapy contrib shares a similar rationale as Django contrib, which is explained in this post. If you are working on a new functionality, please follow that rationale to decide whether it should be a Scrapy contrib. If unsure, you can ask in scrapy-users.

Documentation policies

• Don’t use docstrings for documenting classes, or methods which are already documented in the official (sphinx) documentation. For example, the ItemLoader.add_value() method should be documented in the sphinx documentation, not its docstring.

• Do use docstrings for documenting functions not present in the official (sphinx) documentation, such as functions from scrapy.utils package and its sub-modules.

Tests

Tests are implemented using the Twisted unit-testing framework, running tests requires tox.
Running tests

Make sure you have a recent enough tox installation:

```bash
tox --version
```

If your version is older than 1.7.0, please update it first:

```bash
pip install -U tox
```

To run all tests go to the root directory of Scrapy source code and run:

```bash
tox
```

To run a specific test (say tests/test_loader.py) use:

```bash
 tox -- tests/test_loader.py
```

To see coverage report install coverage (pip install coverage) and run:

```bash
 coverage report
```

see output of coverage --help for more options like html or xml report.

Writing tests

All functionality (including new features and bug fixes) must include a test case to check that it works as expected, so please include tests for your patches if you want them to get accepted sooner.

Scrapy uses unit-tests, which are located in the tests/ directory. Their module name typically resembles the full path of the module they’re testing. For example, the item loaders code is in:

```python
scrapy.loader
```

And their unit-tests are in:

```python
tests/test_loader.py
```

Versioning and API Stability

Versioning

There are 3 numbers in a Scrapy version: \(A.B.C\)

- \(A\) is the major version. This will rarely change and will signify very large changes.
- \(B\) is the release number. This will include many changes including features and things that possibly break backwards compatibility, although we strive to keep theses cases at a minimum.
- \(C\) is the bugfix release number.

Backward-incompatibilities are explicitly mentioned in the release notes, and may require special attention before upgrading.

Development releases do not follow 3-numbers version and are generally released as dev suffixed versions, e.g. 1.3dev.

Note: With Scrapy 0.* series, Scrapy used odd-numbered versions for development releases. This is not the case anymore from Scrapy 1.0 onwards.
Starting with Scrapy 1.0, all releases should be considered production-ready.

For example:

- 1.1.1 is the first bugfix release of the 1.1 series (safe to use in production)

**API Stability**

API stability was one of the major goals for the 1.0 release.

Methods or functions that start with a single dash (\_) are private and should never be relied as stable.

Also, keep in mind that stable doesn’t mean complete: stable APIs could grow new methods or functionality but the existing methods should keep working the same way.

**Release notes**  See what has changed in recent Scrapy versions.

**Contributing to Scrapy**  Learn how to contribute to the Scrapy project.

**Versioning and API Stability**  Understand Scrapy versioning and API stability.
scrapy.contracts, 135
scrapy.contracts.default, 134
scrapy.crawler, 190
scrapy.downloadermiddlewares.ajaxcrawl, 180
scrapy.downloadermiddlewares.chunked, 177
scrapy.downloadermiddlewares.cookies, 170
scrapy.downloadermiddlewares.defaultheaders, 171
scrapy.downloadermiddlewares.downloadtimeout, 172
scrapy.downloadermiddlewares.httpauth, 172
scrapy.downloadermiddlewares.httpcache, 172
scrapy.downloadermiddlewares.httpcompression, 177
scrapy.downloadermiddlewares.httpproxy, 177
scrapy.downloadermiddlewares.redirect, 177
scrapy.downloadermiddlewares.retry, 179
scrapy.downloadermiddlewares.stats, 180
scrapy.downloadermiddlewares.useragent, 180
scrapy.exceptions, 111
scrapy.exporters, 202
scrapy.extensions.closespider, 188
scrapy.extensions.corestats, 187
scrapy.extensions.debug, 189
scrapy.extensions.logstats, 187
scrapy.extensions.memdebug, 188
scrapy.extensions.memusage, 187
scrapy.extensions.statsmailer, 189
scrapy.extensions.telnet, 122
scrapy.http, 79
scrapy.item, 53
scrapy.linkextractors, 89
scrapy.linkextractors.lxmlhtml, 89
scrapy.loader, 57
scrapy.loader.processors, 65
scrapy.mail, 120
scrapy.pipelines.files, 156
scrapy.pipelines.images, 157
scrapy.selector, 50
scrapy.settings, 193
scrapy.signalmanager, 196
scrapy.signals, 198
scrapy.spidermiddlewares, 181
scrapy.spidermiddlewares.depth, 183
scrapy.spidermiddlewares.httperror, 183
scrapy.spidermiddlewares.offsite, 184
scrapy.spidermiddlewares.referer, 184
scrapy.spidermiddlewares.urllength, 185
scrapy.spiders, 31
scrapy.statscollectors, 119
scrapy.utils.log, 118
scrapy.utils.trackref, 150
Symbols
__nonzero__() (scrapy.selector.Selector method), 51
__nonzero__() (scrapy.selector.SelectorList method), 51

A
adapt_response() (scrapy.spiders.XMLFeedSpider method), 38
add_css() (scrapy.loader.ItemLoader method), 62
add_value() (scrapy.loader.ItemLoader method), 61
add_xpath() (scrapy.loader.ItemLoader method), 61
adjust_request_args() (scrapy.contracts.Contract method), 135
AJAXCRAWL_ENABLED setting, 180
AjaxCrawlMiddleware (class in scrapy.downloadermiddlewares.ajaxcrawl), 180
allowed_domains (scrapy.spiders.Spider attribute), 32
AUTOTHROTTLE_DEBUG setting, 161
AUTOTHROTTLE_ENABLED setting, 160
AUTOTHROTTLE_MAX_DELAY setting, 160
AUTOTHROTTLE_START_DELAY setting, 160
AUTOTHROTTLE_TARGET_CONCURRENCY setting, 161
AWS_ACCESS_KEY_ID setting, 93
AWS_SECRET_ACCESS_KEY setting, 93

B
BaseItemExporter (class in scrapy.exporters), 203
BaseSettings (class in scrapy.settings), 193
bench
   command, 30
bindaddress

reqmeta, 84
body (scrapy.http.Request attribute), 81
body (scrapy.http.Response attribute), 87
body_as_unicode() (scrapy.http.TextResponse method), 88
BOT_NAME setting, 93

C
check command, 27
ChunkedTransferMiddleware (class in scrapy.downloadermiddlewares.chunked), 177
clear_stats() (scrapy.statscollectors.StatsCollector method), 197
close_spider(), 71
close_spider() (scrapy.statscollectors.StatsCollector method), 198
closed() (scrapy.spiders.Spider method), 33
CloseSpider, 112
CLOSESPIDER_ERRORCOUNT setting, 189
CLOSESPIDER_ITEMCOUNT setting, 188
CLOSESPIDER_PAGECOUNT setting, 189
CLOSESPIDER_TIMEOUT setting, 188
command
   bench, 30
crawler, 26
edit, 27
fetch, 28
genspider, 26
list, 27
parse, 29
runspider, 30
settings, 29
shell, 28
startproject, 26
version, 30
view, 28
COMMANDS_MODULE
setting, 30
Compose (class in scrapy.loader.processors), 65
COMPRESSION_ENABLED
setting, 177
CONCURRENT_ITEMS
setting, 93
CONCURRENT_REQUESTS
setting, 93
CONCURRENT_REQUESTS_PER_DOMAIN
setting, 93
CONCURRENT_REQUESTS_PER_IP
setting, 93
configure_logging() (in module scrapy.utils.log), 118
connect() (scrapy.signalmanager.SignalManager method), 196
context (scrapy.loader.ItemLoader attribute), 63
Contract (class in scrapy.contracts), 63
cookiejar
reqmeta, 170
COOKIES_DEBUG
setting, 171
COOKIES_ENABLED
setting, 171
CookiesMiddleware (class in scrapy.downloadermiddlewares.cookies), 170
copy() (scrapy.http.Request method), 81
copy() (scrapy.http.Response method), 87
copy() (scrapy.settings.BaseSettings method), 193
copy_to_dict() (scrapy.settings.BaseSettings method), 194
CoreStats (class in scrapy.extensions.corestats), 187
crawl
command, 26
crawl() (scrapy.crawler.Crawler method), 191
crawl() (scrapy.crawler.CrawlerProcess method), 192
crawl() (scrapy.crawler.CrawlerRunner method), 191
Crawler (class in scrapy.crawler), 190
crawler (scrapy.spiders.Spider attribute), 32
CrawlerProcess (class in scrapy.crawler), 192
CrawlerRunner (class in scrapy.crawler), 191
crawlers (scrapy.crawler.CrawlerProcess attribute), 192
crawlers (scrapy.crawler.CrawlerRunner attribute), 191
CrawlSpider (class in scrapy.spiders), 35
create_crawler() (scrapy.crawler.CrawlerProcess method), 192
create_crawler() (scrapy.crawler.CrawlerRunner method), 191
css() (scrapy.http.TextResponse method), 88
css() (scrapy.selector.Selector method), 51
css() (scrapy.selector.SelectorList method), 51
CSVFeedSpider (class in scrapy.spiders), 39
CsvItemExporter (class in scrapy.exporters), 205
custom_settings (scrapy.spiders.Spider attribute), 32
D
default_input_processor (scrapy.loader.ItemLoader attribute), 63
DEFAULT_ITEM_CLASS
setting, 94
default_item_class (scrapy.loader.ItemLoader attribute), 63
default_output_processor (scrapy.loader.ItemLoader attribute), 63
DEFAULT_REQUEST_HEADERS
setting, 94
default_selector_class (scrapy.loader.ItemLoader attribute), 63
DefaultHeadersMiddleware (class in scrapy.downloadermiddlewares.defaultheaders), 171
delimiter (scrapy.spiders.CSVFeedSpider attribute), 39
DEPTH_LIMIT
setting, 94
DEPTH_PRIORITY
setting, 94
DEPTH_STATS
setting, 94
DEPTH_STATS_VERBOSE
setting, 95
DepthMiddleware (class in scrapy.spidermiddlewares.depth), 183
disconnect() (scrapy.signalmanager.SignalManager method), 197
disconnect_all() (scrapy.signalmanager.SignalManager method), 197
DNS_TIMEOUT
setting, 95
DNSCACHE_ENABLED
setting, 95
DNSCACHE_SIZE
setting, 95
dont_cache
reqmeta, 172
dont_obey_robotstxt
reqmeta, 180
dont_redirect
reqmeta, 177
dont_retry
reqmeta, 179
DOWNLOAD_DELAY
setting, 97
DOWNLOAD_HANDLERS
setting, 97
DOWNLOAD_HANDLERS_BASE
setting, 97
download_latency
reqmeta, 84
DOWNLOAD_MAXSIZE
setting, 98
download_maxsize
reqmeta, 98
DOWNLOAD_TIMEOUT
setting, 98
download_timeout
reqmeta, 84
DOWNLOAD_WARNSIZE
setting, 98
DOWNLOADER
setting, 95
DOWNLOADER_CLIENT_TLS_METHOD
setting, 96
DOWNLOADER_CLIENTCONTEXTFACTORY
setting, 95
DOWNLOADER_HTTPCLIENTFACTORY
setting, 95
DOWNLOADER_MIDDLEWARES
setting, 96
DOWNLOADER_MIDDLEWARES_BASE
setting, 96
DOWNLOADER_STATS
setting, 97
DownloaderMiddleware (class in scrapy.downloadermiddlewares), 169
DownloaderStats (class in scrapy.downloadermiddlewares.stats), 180
DownloadTimeoutMiddleware (class in scrapy.downloadermiddlewares.downloadtimeout), 172
DropItem, 111
DummyStatsCollector (class in scrapy.statscollectors), 120
DUPEFILTER_CLASS
setting, 99
DUPEFILTER_DEBUG
setting, 99

text

signal, 199
globalsignal, 199
globalsignal, 199
export_empty_fields (scrapy.exporters.BaseItemExporter attribute), 204
export_item() (scrapy.exporters.BaseItemExporter method), 204
EXTENSIONS
setting, 99
extensions (scrapy.crawler.Crawler attribute), 191
EXTENSIONS_BASE
setting, 99
extract() (scrapy.selector.Selector method), 51
extract() (scrapy.selector.SelectorList method), 51

F
FEED_EXPORT_ENCODING
setting, 78
FEED_EXPORT_FIELDS
setting, 78
FEED_EXPORTERS
setting, 79
FEED_EXPORTERS_BASE
setting, 79
FEED_FORMAT
setting, 78
FEED_STORAGES
setting, 78
FEED_STORAGES_BASE
setting, 78
FEED_STORE_EMPTY
setting, 78
FEED_TEMPDIR
setting, 100
FEED_URI
setting, 78
fetch
command, 28
Field (class in scrapy.item), 56
fields (scrapy.item.Item attribute), 56
fields_to_export (scrapy.exporters.BaseItemExporter attribute), 204
FILES_EXPIRES
setting, 155
FILES_RESULT_FIELD
setting, 154
FILES_STORE
setting, 153
FILES_STORE_S3_ACL
setting, 154
FILES_URLS_FIELD
setting, 154

E
edit
command, 27
EDITOR
setting, 99
encoding (scrapy.exporters.BaseItemExporter attribute), 204
encoding (scrapy.http.TextResponse attribute), 88
eengine (scrapy.crawler.Crawler attribute), 191
FilesPipeline (class in scrapy.pipelines.files), 156
find_by_request() (scrapy.loader.SpiderLoader method), 196
finish_exporting() (scrapy.exporters.BaseItemExporter method), 204
flags (scrapy.http.Response attribute), 87
FormRequest (class in scrapy.http), 84
freeze() (scrapy.settings.BaseSettings method), 194
from_crawler() (scrapy.spiders.Spider method), 32
from_response() (scrapy.http.FormRequest class method), 84
from_settings() (scrapy.loader.SpiderLoader method), 196
from_settings() (scrapy.mail.MailSender class method), 121
frozencopy() (scrapy.settings.BaseSettings method), 194
G
genspider
    command, 26
get() (scrapy.settings.BaseSettings method), 194
get_collected_values() (scrapy.loader.ItemLoader method), 63
get_css() (scrapy.loader.ItemLoader method), 62
get_input_processor() (scrapy.loader.ItemLoader method), 63
get_media_requests() (scrapy.pipelines.files.FilesPipeline method), 156
get_media_requests() (scrapy.pipelines.images.ImagesPipeline method), 156
get_oldest() (in module scrapy.utils.trackref), 151
get_output_processor() (scrapy.loader.ItemLoader method), 63
get_output_value() (scrapy.loader.ItemLoader method), 63
get_settings_priority() (in module scrapy.settings), 193
get_stats() (scrapy.statscollectors.StatsCollector method), 197
get_value() (scrapy.loader.ItemLoader method), 61
get_value() (scrapy.statscollectors.StatsCollector method), 197
get_xpath() (scrapy.loader.ItemLoader method), 61
getbool() (scrapy.settings.BaseSettings method), 194
getdict() (scrapy.settings.BaseSettings method), 194
getfloat() (scrapy.settings.BaseSettings method), 194
getint() (scrapy.settings.BaseSettings method), 194
getlist() (scrapy.settings.BaseSettings method), 195
getpriority() (scrapy.settings.BaseSettings method), 195
getwithbase() (scrapy.settings.BaseSettings method), 195
H
handle_httpstatus_all
    reqmeta, 183
headers (scrapy.http.Request attribute), 81
headers (scrapy.http.Response attribute), 86
headers (scrapy.spiders.CSVFeedSpider attribute), 39
HtmlResponse (class in scrapy.http), 89
HttpAuthMiddleware (class in scrapy.downloadermiddlewares.httpauth), 172
HTTPCACHE_ALWAYS_STORE
    setting, 176
HTTPCACHE_DBM_MODULE
    setting, 175
HTTPCACHE_DIR
    setting, 175
HTTPCACHE_ENABLED
    setting, 174
HTTPCACHE_EXPIRATION_SECS
    setting, 175
HTTPCACHE_GZIP
    setting, 176
HTTPCACHEIGNOREHTTP_CODES
    setting, 175
HTTPCACHEIGNOREMISSING
    setting, 175
HTTPCACHEIGNORERESPONSECACHECONTROLS
    setting, 176
HTTPCACHEIGNORESCHEMES
    setting, 175
HTTPCACHE_POLICY
    setting, 176
HTTPCACHE_STORAGE
    setting, 175
HttpCacheMiddleware (class in scrapy.downloadermiddlewares.httpcache), 172
HttpCompressionMiddleware (class in scrapy.downloadermiddlewares.httpcompression), 177
HTTPERROR_ALLOW_ALL
    setting, 184
HTTPERROR_ALLOWED_CODES
    setting, 184
HttpErrorMiddleware (class in scrapy.spidermiddlewares.httperror), 183
HTTPPROXY_AUTH_ENCODING
    setting, 180
HttpProxyMiddleware (class in scrapy.downloadermiddlewares.httpproxy), 177
I
Identity (class in scrapy.loader.processors), 65
IgnoreRequest, 112
Index

L

list

command, 27
list() (scrapy.loader.SpiderLoader method), 196
load() (scrapy.loader.SpiderLoader method), 196
load_item() (scrapy.loader.ItemLoader method), 62
log() (scrapy.spiders.Spider method), 33

M

MAIL_FROM
setting, 121
MAIL_HOST
setting, 121
MAIL_PASS
setting, 122
MAIL_PORT
setting, 122
MAIL_SSL
setting, 122
MAIL_TLS
setting, 122
MAIL_USER
setting, 122
MailSender (class in scrapy.mail), 120
make_requests_from_url() (scrapy.spiders.Spider method), 33
MapCompose (class in scrapy.loader.processors), 66
max_value() (scrapy.statscollectors.StatsCollector method), 197
maxpriority() (scrapy.settings.BaseSettings method), 195
MEMDEBUG_ENABLED
setting, 101
MEMDEBUG_NOTIFY
setting, 101
MemoryStatsCollector (class in scrapy.statscollectors), 120
MEMUSAGE_CHECK_INTERVAL_SECONDS
setting, 102
MEMUSAGE_ENABLED
setting, 101
MEMUSAGE_LIMIT_MB
setting, 101
MEMUSAGE_NOTIFY_MAIL

J

Join (class in scrapy.loader.processors), 65
join() (scrapy.crawler.CrawlerProcess method), 192
join() (scrapy.crawler.CrawlerRunner method), 192
JsonItemExporter (class in scrapy.exporters), 206
JsonLinesItemExporter (class in scrapy.exporters), 207

I

inc_value() (scrapy.statscollectors.StatsCollector method), 197
Item (class in scrapy.item), 56
item (scrapy.loader.ItemLoader attribute), 63
item_completed() (scrapy.pipelines.files.FilesPipeline method), 157
item_completed() (scrapy.pipelines.images.ImagesPipeline method), 158
item_dropped
signal, 199
item_dropped() (in module scrapy.signals), 199
ITEM_PIPELINES
setting, 100
ITEM_PIPELINES_BASE
setting, 100
item_scraped
signal, 199
item_scraped() (in module scrapy.signals), 199
ItemLoader (class in scrapy.loader), 60
iter_all() (in module scrapy.utils.trackref), 151
iterator (scrapy.spiders.XMLFeedSpider attribute), 37
itertag (scrapy.spiders.XMLFeedSpider attribute), 37

Scrapy Documentation, Release 1.2.3
setting, 102
MEMUSAGE_REPORT
setting, 102
MEMUSAGE_WARNING_MB
setting, 102
meta (scrapy.http.Request attribute), 81
meta (scrapy.http.Response attribute), 87
METAREFRESH_ENABLED
setting, 178
METAREFRESH_MAXDELAY
setting, 178
MetaRefreshMiddleware (class in scrapy.downloadermiddlewares.redirect), 178
method (scrapy.http.Request attribute), 81
min_value() (scrapy.statscollectors.StatsCollector method), 197

name (scrapy.spiders.Spider attribute), 32
namespaces (scrapy.spiders.XMLFeedSpider attribute), 37
nested_css() (scrapy.loader.ItemLoader method), 62
nested_xpath() (scrapy.loader.ItemLoader method), 62
NEWSPIDER_MODULE
setting, 102
NotConfigured, 112
NotSupported, 112

object_ref (class in scrapy.utils.trackref), 150
OffsiteMiddleware (class in scrapy.spidermiddlewares.offsite), 184
open_spider(), 71
open_spider() (scrapy.statscollectors.StatsCollector method), 197

parse
command, 29
parse() (scrapy.spiders.Spider method), 33
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
PrintItemExporter (class in scrapy.exporters), 206
pre_process() (scrapy.contracts.Contract method), 135
print_live_refs() (in module scrapy.utils.trackref), 150
process_exception() (scrapy.downloadermiddlewares.DownloaderMiddleware method), 169
process_request() (scrapy.downloadermiddlewares.DownloaderMiddleware method), 169
process_response() (scrapy.downloadermiddlewares.DownloaderMiddleware method), 169
process_results() (scrapy.spiders.XMLFeedSpider method), 38
process_spider_exception() (scrapy.spidermiddlewares.SpiderMiddleware method), 182
process_spider_input() (scrapy.spidermiddlewares.SpiderMiddleware method), 181
process_spider_output() (scrapy.spidermiddlewares.SpiderMiddleware method), 182
process_start_requests() (scrapy.spidermiddlewares.SpiderMiddleware method), 182
proxy
reqmeta, 177
Python Enhancement Proposals
PEP 8, 248

quotechar (scrapy.spiders.CSVFeedSpider attribute), 39

RANDOMIZE_DOWNLOAD_DELAY
setting, 103
re() (scrapy.selector.Selector method), 51
re() (scrapy.selector.SelectorList method), 51
REACTOR_THREADPOOL_MAXSIZE
setting, 103
REDIRECT_ENABLED
setting, 178
REDIRECT_MAX_TIMES
setting, 103, 178
REDIRECT_PRIORITY_ADJUST
setting, 103
redirect_urls
reqmeta, 177
RedirectMiddleware (class in scrapy.downloadermiddlewares.redirect), 177
REFERER_ENABLED
setting, 184
RefererMiddleware (class in scrapy.spidermiddlewaresreferer), 184
register_namespace() (scrapy.selector.Selector method), 51
remove_namespaces() (scrapy.selector.Selector method), 51
replace() (scrapy.http.Request method), 81
replace() (scrapy.http.Response method), 87
replace_css() (scrapy.loader.ItemLoader method), 62
replace_value() (scrapy.loader.ItemLoader method), 63
replace_xpath() (scrapy.loader.ItemLoader method), 62
reqmeta
bindaddress, 84
cookiejar, 170
don't_cache, 172
don't_obey_robotstxt, 180
don't_redirect, 177
don't_retry, 179
download_latency, 84
download_maxsize, 98
download_timeout, 84
handle_httpstatus_all, 183
handle_httpstatus_list, 183
proxy, 177
redirect_urls, 177
Request (class in scrapy.http), 80
request (scrapy.http.Response attribute), 87
request_dropped
signal, 201
request_dropped() (in module scrapy.signals), 201
requestScheduled
signal, 201
requestScheduled() (in module scrapy.signals), 201
Response (class in scrapy.http), 86
response_downloaded
signal, 201
response_downloaded() (in module scrapy.signals), 201
responseReceived
signal, 201
responseReceived() (in module scrapy.signals), 201
RETRY_ENABLED
setting, 179
RETRY_HTTP_CODES
setting, 179
RETRY_PRIORITY_ADJUST
setting, 103
RETRY_TIMES
setting, 179
RetryMiddleware (class in scrapy.downloadermiddlewares.retry), 179
ReturnsContract (class in scrapycontracts.default), 134
ROBOTSTXT_OBEY
setting, 104
RobotsTxtMiddleware (class in scrapy.downloadermiddlewares.robotstxt), 179
Rule (class in scrapy.spiders), 36
rules (scrapy.spiders.CrawlSpider attribute), 35
runspider
command, 30
S
SCHEDULER
setting, 104
SCHEDULER_DEBUG
setting, 104
SCHEDULER_DISK_QUEUE
setting, 104
SCHEDULER_MEMORY_QUEUE
setting, 104
SCHEDULER_PRIORITY_QUEUE
setting, 105
ScrapesContract (class in scrapycontracts.default), 134
scrapy.contracts (module), 135
scrapy.contracts.default (module), 134
scrapy.crawler (module), 190
scrapy.downloadermiddlewares (module), 169
scrapy.downloadermiddlewares.ajaxcrawl (module), 180
scrapy.downloadermiddlewares.chunked (module), 177
scrapy.downloadermiddlewares.cookies (module), 170
scrapy.downloadermiddlewares.defaultheaders (module), 171
scrapy.downloadermiddlewares.downloadtimeout (module), 172
scrapy.downloadermiddlewares.httpproxy (module), 177
scrapy.downloadermiddlewares.retry (module), 179
scrapy.downloadermiddlewares.redirect (module), 177
scrapy.downloadermiddlewares.stats (module), 180
scrapy.downloadermiddlewares.useragent (module), 180
scrapy.exceptions (module), 111
scrapy.exporters (module), 202
scrapy.extensions.closespider (module), 188
scrapy.extensions.closespider.CloseSpider (class in scrapy.extensions.closespider), 188
scrapy.extensions.corestats (module), 187
scrapy.extensions.debug (module), 189
scrapy.extensions.debug.Debugger (class in scrapy.extensions.debug), 190
scrapy.extensions.debug.StackTraceDump (class in scrapy.extensions.debug), 189
scrapy.extensions.statsmailer (module), 189
scrapy.extensions.statsmailer.StatsMailer (class in scrapy.extensions.statsmailer), 189
scrapy.extensions.telnet (module), 187
scrapy.extensions.telnet.TelnetConsole (class in scrapy.extensions.telnet), 187
scrapy.http (module), 79
scrapy.item (module), 53
scrapy.linkextractors (module), 89
scrapy.linkextractors.lxmlhtml (module), 89
scrapy.loader (module), 57, 196
scrapy.loader.processors (module), 65
scrapy.mail (module), 120
scrapy.pipelines.files (module), 156
scrapy.pipelines.images (module), 157
scrapy.selector (module), 50
scrapy.settings (module), 193
scrapy.signalmanager (module), 196
scrapy.signals (module), 198
scrapy.spidermiddlewares (module), 181
scrapy.spidermiddlewares.depth (module), 183
scrapy.spidermiddlewares.httperror (module), 183
scrapy.spidermiddlewares.offsite (module), 184
scrapy.spidermiddlewares.referer (module), 184
scrapy.spidermiddlewares.urllength (module), 185
scrapy.spiders (module), 31
scrapy.statscollectors (module), 119, 197
scrapy.utils.log (module), 118
scrapy.utils.trackref (module), 150
SelectJmes (class in scrapy.loader.processors), 66
Selector (class in scrapy.selector), 50
selector (scrapy.http.TextResponse attribute), 88
SelectorList (class in scrapy.selector), 51
send() (scrapy.mail.MailSender method), 121
send_catch_log() (scrapy.signalmanager.SignalManager method), 197
send_catch_log_deferred() (scrapy.signalmanager.SignalManager method), 197
serialize_field() (scrapy.exporters.BaseItemExporter method), 204
set() (scrapy.settings.BaseSettings method), 204
set_stats() (scrapy.statscollectors.StatsCollector method), 197
set_value() (scrapy.statscollectors.StatsCollector method), 197
setmodule() (scrapy.settings.BaseSettings method), 195
AJAXCRAWL_ENABLED, 180
AUTOTHROTTLE_DEBUG, 161
AUTOTHROTTLE_ENABLED, 160
AUTOTHROTTLE_MAX_DELAY, 160
AUTOTHROTTLE_START_DELAY, 160
AUTOTHROTTLE_TARGET_CONCURRENCY, 161
AWS_ACCESS_KEY_ID, 93
AWS_SECRET_ACCESS_KEY, 93
BOT_NAME, 93
CLOSESPIDER_ERRORCOUNT, 189
CLOSESPIDER_ITEMCOUNT, 188
CLOSESPIDER_PAGECOUNT, 189
CLOSESPIDER_TIMEOUT, 188
COMMANDS_MODULE, 30
COMPRESSION_ENABLED, 177
CONCURRENT_ITEMS, 93
CONCURRENT_REQUESTS, 93
CONCURRENT_REQUESTS_PER_DOMAIN, 93
CONCURRENT_REQUESTS_PER_IP, 93
COOKIES_DEBUG, 171
COOKIES_ENABLED, 171
DEFAULT_ITEM_CLASS, 94
DEFAULT_REQUEST_HEADERS, 94
DEPTH_LIMIT, 94
DEPTH_PRIORITY, 94
DEPTH_STATS, 94
DOWNLOAD_DELAY, 97
DOWNLOAD_HANDLERS, 97
DOWNLOAD_HANDLERS_BASE, 97
DOWNLOAD_MAXSIZE, 98
DOWNLOAD_TIMEOUT, 98
DOWNLOAD_WARNSIZE, 98
DOWNLOADER, 95
DOWNLOADER_CLIENT_TLS_METHOD, 96
DOWNLOADER_CLIENTCONTEXTFACTORY, 95
DOWNLOADER_HTTPCLIENTFACTORY, 95
DOWNLOADER_MIDDLEWARES, 96
DOWNLOADER_MIDDLEWARES_BASE, 96
DOWNLOADER_STATS, 97
DUPEFILTER_CLASS, 99
DUPEFILTER_DEBUG, 99
EDITOR, 99
EXTENSIONS, 99
EXTENSIONS_BASE, 99
FEED_EXPORT_ENCODING, 78
FEED_EXPORT_FIELDS, 78
FEED_EXPORTERS, 79
FEED_EXPORTERS_BASE, 79
FEED_FORMAT, 78
FEED_STORAGES, 78
FEED_STORAGES_BASE, 78
FEED_STORE_EMPTY, 78
FEED_TEMPDIR, 100
FEED_URI, 78
FILES_EXPIRES, 155
FILES_RESULT_FIELD, 154
FILES_STORE, 153
FILES_STORE_S3_ACL, 154
FILES_URLS_FIELD, 154
HTTPCACHE_ALWAYS_STORE, 176
HTTPCACHE_DBM_MODULE, 175
HTTPCACHE_DIR, 175
HTTPCACHE_ENABLED, 174
HTTPCACHE_EXPIRATION_SECS, 175
HTTPCACHE_GZIP, 176
HTTPCACHE_IGNORE_HTTP_CODES, 175
HTTPCACHE_IGNORE_MISSING, 175
HTTPCACHE_IGNORE_RESPONSE_CACHE_CONTROLS, 176
HTTPCACHE_POLICY, 176
HTTPCACHE_STORAGE, 175
HTTPERROR_ALLOW_ALL, 184
HTTPERROR_ALLOWED_CODES, 184
HTTPPROXY_AUTH_ENCODING, 180
IMAGES_EXPIRES, 155
IMAGES_MIN_HEIGHT, 156
IMAGES_MIN_WIDTH, 156
IMAGES_RESULT_FIELD, 154
IMAGES_STORE, 153
IMAGES_STORE_S3_ACL, 154
IMAGES_THUMBS, 155
IMAGES_URLS_FIELD, 154
ITEM_PIPELINES, 100
ITEM_PIPELINES_BASE, 100
LOG_DATEFORMAT, 101
LOG_DISABLED, 100
LOG_ENCODING, 100
LOG_FILE, 100
LOG_FORMAT, 100
LOG_LEVEL, 101
LOG_STDOUT, 101
MAIL_FROM, 121
MAIL_HOST, 121
MAIL_PASS, 122
MAIL_PORT, 122
MAIL_SSL, 122
MAIL_TLS, 122
MAIL_USER, 122
MEMDEBUG_ENABLED, 101
MEMDEBUG_NOTIFY, 101
MEMUSAGE_CHECK_INTERVAL SECONDS, 102
MEMUSAGE_ENABLED, 101
MEMUSAGE_LIMIT_MB, 101
MEMUSAGE_NOTIFY_MAIL, 102
MEMUSAGE_REPORT, 102
MEMUSAGE_WARNING_MB, 102
METAREFRESH_ENABLED, 178
METAREFRESH_MAXDELAY, 178
NEWSPIDER_MODULE, 102
RANDOMIZE_DOWNLOAD_DELAY, 103
REACTOR_THREADPOOL_MAXSIZE, 103
REDIRECT_MAX_TIMES, 103, 178
REDIRECT_PRIORITY_ADJUST, 103
REFERER_ENABLED, 184
RETRY_ENABLED, 179
RETRY_HTTP_CODES, 179
RETRY_PRIORITY_ADJUST, 103
RETRY_TIMES, 179
SPIDER_CONTRACTS, 105
SPIDER_CONTRACTS_BASE, 105
SPIDER_MIDDLEWARES, 105
SPIDER_MIDDLEWARES_BASE, 105
SPIDER_MODULES, 106
STATS_CLASS, 106
STATS_DUMP, 106
STATSMAILER_RCPTS, 106
TELNETCONSOLE_ENABLED, 106
TELNETCONSOLE_HOST, 124
TELNETCONSOLE_PORT, 106, 124
TEMPLATES_DIR, 107
URLLENGTH_LIMIT, 107
USER_AGENT, 107
settings
command, 29
Settings (class in scrapy.settings), 193
settings (scrapy.crawler.Crawler attribute), 190
settings (scrapy.spiders.Spider attribute), 32
SETTINGS_PRIORITIES (in module scrapy.settings), 193
shell
command, 28
signal
engine_started, 199
engine_stopped, 199
item_dropped, 199
item_scraped, 199
request_dropped, 201
request_scheduled, 201
response_downloaded, 201
response_received, 201
spider_closed, 199
spider_error, 200
spider_idle, 200
spider_opened, 200
update_telnets vars, 124
SignalManager (class in scrapy.signalmanager), 196
signals (scrapy.crawler.Crawler attribute), 190