

---

# Scrapy Documentation

*Release 0.14.4*

**Insophia**

January 23, 2014



<b>1</b>	<b>Getting help</b>	<b>3</b>
<b>2</b>	<b>First steps</b>	<b>5</b>
2.1	Scrapy at a glance . . . . .	5
2.2	Installation guide . . . . .	8
2.3	Scrapy Tutorial . . . . .	11
2.4	Examples . . . . .	17
<b>3</b>	<b>Basic concepts</b>	<b>19</b>
3.1	Command line tool . . . . .	19
3.2	Items . . . . .	26
3.3	Spiders . . . . .	29
3.4	Link Extractors . . . . .	37
3.5	XPath Selectors . . . . .	39
3.6	Item Loaders . . . . .	45
3.7	Scrapy shell . . . . .	52
3.8	Item Pipeline . . . . .	55
3.9	Feed exports . . . . .	58
3.10	Link Extractors . . . . .	62
<b>4</b>	<b>Built-in services</b>	<b>65</b>
4.1	Logging . . . . .	65
4.2	Stats Collection . . . . .	67
4.3	Sending e-mail . . . . .	70
4.4	Telnet Console . . . . .	72
4.5	Web Service . . . . .	74
<b>5</b>	<b>Solving specific problems</b>	<b>81</b>
5.1	Frequently Asked Questions . . . . .	81
5.2	Using Firefox for scraping . . . . .	85
5.3	Using Firebug for scraping . . . . .	86
5.4	Debugging memory leaks . . . . .	90
5.5	Downloading Item Images . . . . .	94
5.6	Ubuntu packages . . . . .	99
5.7	Scrapy Service (scrapyd) . . . . .	99
5.8	Jobs: pausing and resuming crawls . . . . .	108
<b>6</b>	<b>Extending Scrapy</b>	<b>111</b>
6.1	Architecture overview . . . . .	111

6.2	Downloader Middleware . . . . .	114
6.3	Spider Middleware . . . . .	121
6.4	Extensions . . . . .	124
<b>7</b>	<b>Reference</b>	<b>131</b>
7.1	Requests and Responses . . . . .	131
7.2	Settings . . . . .	138
7.3	Signals . . . . .	151
7.4	Exceptions . . . . .	154
7.5	Item Exporters . . . . .	155
<b>8</b>	<b>All the rest</b>	<b>161</b>
8.1	Contributing to Scrapy . . . . .	161
8.2	Versioning and API Stability . . . . .	163
8.3	Experimental features . . . . .	164
	<b>Python Module Index</b>	<b>167</b>

This documentation contains everything you need to know about Scrapy.



---

### 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*.
- 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](#).





---

## First steps

---

### 2.1 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 [screen scraping](#) (more precisely, [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.

The purpose of this document is to introduce you to the concepts behind Scrapy so you can get an idea of how it works and decide if Scrapy is what you need.

When you're ready to start a project, you can *[start with the tutorial](#)*.

#### 2.1.1 Pick a website

So you need to extract some information from a website, but the website doesn't provide any API or mechanism to access that info programmatically. Scrapy can help you extract that information.

Let's say we want to extract the URL, name, description and size of all torrent files added today in the [Mininova](#) site.

The list of all torrents added today can be found on this page:

<http://www.mininova.org/today>

#### 2.1.2 Define the data you want to scrape

The first thing is to define the data we want to scrape. In Scrapy, this is done through *Scrapy Items* (Torrent files, in this case).

This would be our Item:

```
from scrapy.item import Item, Field

class Torrent(Item):
    url = Field()
    name = Field()
    description = Field()
    size = Field()
```

### 2.1.3 Write a Spider to extract the data

The next thing is to write a Spider which defines the start URL (<http://www.mininova.org/today>), the rules for following links and the rules for extracting the data from pages.

If we take a look at that page content we'll see that all torrent URLs are like <http://www.mininova.org/tor/NUMBER> where NUMBER is an integer. We'll use that to construct the regular expression for the links to follow: `/tor/\d+`.

We'll use [XPath](#) for selecting the data to extract from the web page HTML source. Let's take one of those torrent pages:

<http://www.mininova.org/tor/2657665>

And look at the page HTML source to construct the XPath to select the data we want which is: torrent name, description and size.

By looking at the page HTML source we can see that the file name is contained inside a `<h1>` tag:

```
<h1>Home [2009] [Eng] XviD-ovd</h1>
```

An XPath expression to extract the name could be:

```
//h1/text()
```

And the description is contained inside a `<div>` tag with `id="description"`:

```
<h2>Description:</h2>
```

```
<div id="description">
```

```
"HOME" - a documentary film by Yann Arthus-Bertrand
```

```
<br/>
```

```
<br/>
```

```
***
```

```
<br/>
```

```
<br/>
```

```
"We are living in exceptional times. Scientists tell us that we have 10 years to change the way we live."
```

```
...
```

An XPath expression to select the description could be:

```
//div[@id='description']
```

Finally, the file size is contained in the second `<p>` tag inside the `<div>` tag with `id=specifications`:

```
<div id="specifications">
```

```
<p>
```

```
<strong>Category:</strong>
```

```
<a href="/cat/4">Movies</a> &gt; <a href="/sub/35">Documentary</a>
```

```
</p>
```

```
<p>
```

```
<strong>Total size:</strong>
```

```
699.79&nbsp;megabyte</p>
```

An XPath expression to select the description could be:

```
//div[@id='specifications']/p[2]/text()[2]
```

For more information about XPath see the [XPath reference](#).

Finally, here's the spider code:

```
class MininovaSpider(CrawlSpider):

    name = 'mininova.org'
    allowed_domains = ['mininova.org']
    start_urls = ['http://www.mininova.org/today']
    rules = [Rule(SgmlLinkExtractor(allow=['/tor/\d+']), 'parse_torrent')]

    def parse_torrent(self, response):
        x = HtmlXPathSelector(response)

        torrent = TorrentItem()
        torrent['url'] = response.url
        torrent['name'] = x.select("//h1/text()").extract()
        torrent['description'] = x.select("//div[@id='description']").extract()
        torrent['size'] = x.select("//div[@id='info-left']/p[2]/text()[2]").extract()
        return torrent
```

For brevity's sake, we intentionally left out the import statements. The `Torrent` item is *defined above*.

## 2.1.4 Run the spider to extract the data

Finally, we'll run the spider to crawl the site and output a file `scraped_data.json` with the scraped data in JSON format:

```
scrapy crawl mininova.org -o scraped_data.json -t json
```

This uses *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 very easily.

## 2.1.5 Review scraped data

If you check the `scraped_data.json` file after the process finishes, you'll see the scraped items there:

```
[{"url": "http://www.mininova.org/tor/2657665", "name": ["Home[2009][Eng]XviD-ovd"], "description":
# ... other items ...
}]
```

You'll notice that all field values (except for the `url` which was assigned directly) are actually lists. This is because the *selectors* return lists. You may want to store single values, or perform some additional parsing/cleansing to the values. That's what *Item Loaders* are for.

## 2.1.6 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 and XML sources
- Built-in support for cleaning and sanitizing the scraped data using a collection of reusable filters (called *Item Loaders*) shared between all the spiders.
- Built-in support for *generating feed exports* in multiple formats (JSON, CSV, XML) and storing them in multiple backends (FTP, S3, local filesystem)

- A media pipeline for *automatically downloading images* (or any other media) associated with the scraped items
- Support for *extending Scrapy* by plugging your own functionality using *signals* and a well-defined API (middlewares, *extensions*, and *pipelines*).
- Wide range of built-in middlewares and extensions for:
  - cookies and session handling
  - HTTP compression
  - HTTP authentication
  - HTTP cache
  - user-agent spoofing
  - robots.txt
  - crawl depth restriction
  - and more
- Robust encoding support and auto-detection, for dealing with foreign, non-standard and broken encoding declarations.
- Extensible *stats collection* for multiple spider metrics, useful for monitoring the performance of your spiders and detecting when they get broken
- An *Interactive shell console* for trying XPaths, very useful for writing and debugging your spiders
- A *System service* designed to ease the deployment and run of your spiders in production.
- A built-in *Web service* for monitoring and controlling your bot
- A *Telnet console* for hooking into a Python console running inside your Scrapy process, to introspect and debug your crawler
- *Logging* facility that you can hook on to for catching errors during the scraping process.
- Support for crawling based on URLs discovered through *Sitemaps*
- A caching DNS resolver

### 2.1.7 What's next?

The next obvious steps are for you to [download Scrapy](#), read [the tutorial](#) and join [the community](#). Thanks for your interest!

## 2.2 Installation guide

This document describes how to install Scrapy on Linux, Windows and Mac OS X.

### 2.2.1 Requirements

- [Python 2.5, 2.6, 2.7](#) (3.x is not yet supported)
- [Twisted 2.5.0, 8.0](#) or above (Windows users: you'll need to install [Zope.Interface](#) and maybe [pywin32](#) because of this [Twisted bug](#))
- [w3lib](#)

- `lxml` or `libxml2` (if using `libxml2`, version 2.6.28 or above is highly recommended)
- `simplejson` (not required if using Python 2.6 or above)
- `pyopenssl` (for HTTPS support. Optional, but highly recommended)

## 2.2.2 Install Python

First, you need to install Python, if you haven't done so already.

Scrapy works with Python 2.5, 2.6 or 2.7, which you can get at <http://www.python.org/download/>

**See also:**

*What Python versions does Scrapy support?*

## 2.2.3 Install Scrapy

There are many ways to install Scrapy. Pick the one you feel more comfortable with.

- *Download and install an official release* (requires installing dependencies separately)
- *Installing with `easy_install`* (automatically installs dependencies)
- *Installing with `pip`* (automatically installs dependencies)

### Download and install an official release

Download Scrapy from the [Download page](#). Scrapy is distributed in two ways: a source code tarball (for Unix and Mac OS X systems) and a Windows installer (for Windows). If you downloaded the tarball, you can install it as any Python package using `setup.py`:

```
tar xzf Scrapy-X.X.X.tar.gz
cd Scrapy-X.X.X
python setup.py install
```

If you downloaded the Windows installer, just run it.

**Warning:** In Windows, you may need to add the `C:\Python25\Scripts` (or `C:\Python26\Scripts`) folder to the system path by adding that directory to the `PATH` environment variable from the [Control Panel](#).

### Installing with `easy_install`

You can install Scrapy using `setuptools`'s `easy_install` with:

```
easy_install -U Scrapy
```

### Installing with `pip`

You can install Scrapy using `pip` with:

```
pip install Scrapy
```

## 2.2.4 Platform specific instructions

### Linux

#### Ubuntu 9.10 or above

If you're running Ubuntu 9.10 (or above), use the official [Ubuntu Packages](#), which already solve all dependencies for you and are continuously updated with the latest bug fixes.

#### Debian or Ubuntu (9.04 or older)

If you're running Debian Linux, run the following command as root:

```
apt-get install python-twisted python-libxml2 python-pyopenssl python-simplejson
```

Then:

```
easy_install -U w3lib
```

And then follow the instructions in [Install Scrapy](#).

#### Arch Linux

If you are running Arch Linux, run the following command as root:

```
pacman -S twisted libxml2 pyopenssl python-simplejson
```

Then:

```
easy_install -U w3lib
```

And then follow the instructions in [Install Scrapy](#).

#### Other Linux distros

The easiest way to install Scrapy on other Linux distros is through `easy_install` or `pip`, which will automatically install Twisted, w3lib and lxml dependencies.

See [Installing with easy\\_install](#) or [Installing with pip](#)

Another way would be to install dependencies, if you know the packages in your distros that meets them. See [Requirements](#).

### Mac OS X

The easiest way to install Scrapy on Mac is through `easy_install` or `pip`, which will automatically install Twisted, w3lib and lxml dependencies.

See [Installing with easy\\_install](#) or [Installing with pip](#)

## Windows

The easiest way to install Scrapy on Windows is through `easy_install` or `pip`, which will automatically install Twisted, w3lib and lxml dependencies.

See [Installing with easy\\_install](#) or [Installing with pip](#)

## 2.3 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 use [Open directory project \(dmoz\)](#) as our example domain to scrape.

This tutorial will walk you through these tasks:

1. Creating a new Scrapy project
2. Defining the Items you will extract
3. Writing a *spider* to crawl a site and extract *Items*
4. Writing an *Item Pipeline* to store the extracted Items

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 [Dive Into Python](#). If you're new to programming and want to start with Python, take a look at [this list of Python resources for non-programmers](#).

### 2.3.1 Creating a project

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

```
scrapy startproject tutorial
```

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

```
tutorial/  
  scrapy.cfg  
  tutorial/  
    __init__.py  
    items.py  
    pipelines.py  
    settings.py  
    spiders/  
      __init__.py  
      ...
```

These are basically:

- `scrapy.cfg`: the project configuration file
- `tutorial/`: the project's python module, you'll later import your code from here.
- `tutorial/items.py`: the project's items file.
- `tutorial/pipelines.py`: the project's pipelines file.
- `tutorial/settings.py`: the project's settings file.

- `tutorial/spiders/`: a directory where you'll later put your spiders.

### 2.3.2 Defining our Item

*Items* are containers that will be loaded with the scraped data; they work like simple python dicts but provide additional protecting against populating undeclared fields, to prevent typos.

They are declared by creating an `scrapy.item.Item` class and defining its attributes as `scrapy.item.Field` objects, like you will in an ORM (don't worry if you're not familiar with ORMs, you will see that this is an easy task).

We begin by modeling the item that we will use to hold the sites data obtained from `dmoz.org`, as we want to capture the name, url and description of the sites, we define fields for each of these three attributes. To do that, we edit `items.py`, found in the `tutorial` directory. Our Item class looks like this:

```
from scrapy.item import Item, Field

class DmozItem(Item):
    title = Field()
    link = Field()
    desc = Field()
```

This may seem complicated at first, but defining the item allows you to use other handy components of Scrapy that need to know how your item looks like.

### 2.3.3 Our first Spider

Spiders are user-written classes used to scrape information from a domain (or group of domains).

They define an initial list of URLs to download, how to follow links, and how to parse the contents of those pages to extract *items*.

To create a Spider, you must subclass `scrapy.spider.BaseSpider`, and define the three main, mandatory, attributes:

- `name`: identifies the Spider. It must be unique, that is, you can't set the same name for different Spiders.
- `start_urls`: is a list of URLs where the Spider will begin to crawl from. So, the first pages downloaded will be those listed here. The subsequent URLs will be generated successively from data contained in the start URLs.
- `parse()` is a method of the spider, which will be called with the downloaded `Response` object of each start URL. The response is passed to the method as the first and only argument.

This method is responsible for parsing the response data and extracting scraped data (as scraped items) and more URLs to follow.

The `parse()` method is in charge of processing the response and returning scraped data (as `Item` objects) and more URLs to follow (as `Request` objects).

This is the code for our first Spider; save it in a file named `dmoz_spider.py` under the `dmoz/spiders` directory:

```
from scrapy.spider import BaseSpider

class DmozSpider(BaseSpider):
    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/"
```



```
]

def parse(self, response):
    filename = response.url.split("/")[-2]
    open(filename, 'wb').write(response.body)
```

## Crawling

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

```
scrapy crawl dmoz
```

The `crawl dmoz` command runs the spider for the `dmoz.org` domain. You will get an output similar to this:

```
2008-08-20 03:51:13-0300 [scrapy] INFO: Started project: dmoz
2008-08-20 03:51:13-0300 [tutorial] INFO: Enabled extensions: ...
2008-08-20 03:51:13-0300 [tutorial] INFO: Enabled downloader middlewares: ...
2008-08-20 03:51:13-0300 [tutorial] INFO: Enabled spider middlewares: ...
2008-08-20 03:51:13-0300 [tutorial] INFO: Enabled item pipelines: ...
2008-08-20 03:51:14-0300 [dmoz] INFO: Spider opened
2008-08-20 03:51:14-0300 [dmoz] DEBUG: Crawled <http://www.dmoz.org/Computers/Programming/Languages/Python/Books>
2008-08-20 03:51:14-0300 [dmoz] DEBUG: Crawled <http://www.dmoz.org/Computers/Programming/Languages/Python/Resources>
2008-08-20 03:51:14-0300 [dmoz] INFO: Spider closed (finished)
```

Pay attention to the lines containing `[dmoz]`, which corresponds to our spider. You can see a log line for each URL defined in `start_urls`. Because these URLs are the starting ones, they have no referrers, which is shown at the end of the log line, where it says `(referer: <None>)`.

But more interesting, as our `parse` method instructs, two files have been created: *Books* and *Resources*, with the content of both URLs.

## What just happened under the hood?

Scrapy creates `scrapy.http.Request` objects for each URL in the `start_urls` attribute of the Spider, and assigns them the `parse` method of the spider as their callback function.

These Requests are scheduled, then executed, and `scrapy.http.Response` objects are returned and then fed back to the spider, through the `parse()` method.

## Extracting Items

### Introduction to Selectors

There are several ways to extract data from web pages. Scrapy uses a mechanism based on [XPath](#) expressions called *XPath selectors*. For more information about selectors and other extraction mechanisms see the *XPath selectors documentation*.

Here are some examples of XPath expressions and their meanings:

- `/html/head/title`: selects the `<title>` element, inside the `<head>` element of a HTML document
- `/html/head/title/text()`: selects the text inside the aforementioned `<title>` element.
- `//td`: selects all the `<td>` elements
- `//div[@class="mine"]`: selects all `div` elements which contain an attribute `class="mine"`

These are just a couple of simple examples of what you can do with XPath, but XPath expressions are indeed much more powerful. To learn more about XPath we recommend [this XPath tutorial](#).

For working with XPaths, Scrapy provides a `XPathSelector` class, which comes in two flavours, `HtmlXPathSelector` (for HTML data) and `XmlXPathSelector` (for XML data). In order to use them you must instantiate the desired class with a `Response` object.

You can see selectors as objects that represent nodes in the document structure. So, the first instantiated selectors are associated to the root node, or the entire document.

Selectors have three methods (click on the method to see the complete API documentation).

- `select()`: returns a list of selectors, each of them representing the nodes selected by the xpath expression given as argument.
- `extract()`: returns a unicode string with the data selected by the XPath selector.
- `re()`: returns a list of unicode strings extracted by applying the regular expression given as argument.

### Trying Selectors in the Shell

To illustrate the use of Selectors we're going to use the built-in *Scrapy shell*, which also requires IPython (an extended Python console) installed on your system.

To start a shell, you must go to the project's top level directory and run:

```
scrapy shell http://www.dmoz.org/Computers/Programming/Languages/Python/Books/
```

This is what the shell looks like:

```
[ ... Scrapy log here ... ]
```

```
[s] Available Scrapy objects:
[s] 2010-08-19 21:45:59-0300 [default] INFO: Spider closed (finished)
[s] hxs      <HtmlXPathSelector (http://www.dmoz.org/Computers/Programming/Languages/Python/Books/)>
[s] item      Item()
[s] request   <GET http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
[s] response  <200 http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
[s] spider    <BaseSpider 'default' at 0x1b6c2d0>
[s] xxs      <XmlXPathSelector (http://www.dmoz.org/Computers/Programming/Languages/Python/Books/)>
[s] Useful shortcuts:
[s] shelp()      Print this help
[s] fetch(req_or_url) Fetch a new request or URL and update shell objects
[s] view(response) View response in a browser
```

```
In [1]:
```

After the shell loads, you will have the response fetched in a local `response` variable, so if you type `response.body` you will see the body of the response, or you can type `response.headers` to see its headers.

The shell also instantiates two selectors, one for HTML (in the `hxs` variable) and one for XML (in the `xxs` variable) with this response. So let's try them:

```
In [1]: hxs.select('//title')
Out[1]: [<HtmlXPathSelector (title) xpath=//title>]

In [2]: hxs.select('//title').extract()
Out[2]: [u'<title>Open Directory - Computers: Programming: Languages: Python: Books</title>']
```

```
In [3]: hxs.select('//title/text()')
Out[3]: [<HtmlXPathSelector (text) xpath=//title/text(>]

In [4]: hxs.select('//title/text()').extract()
Out[4]: [u'Open Directory - Computers: Programming: Languages: Python: Books']

In [5]: hxs.select('//title/text()').re('(\w+):')
Out[5]: [u'Computers', u'Programming', u'Languages', u'Python']
```

## Extracting the data

Now, let's try to extract some real information from those pages.

You could type `response.body` in the console, and inspect the source code to figure out the XPath paths you need to use. However, inspecting the raw HTML code there could become a very tedious task. To make this an easier task, you can use some Firefox extensions like Firebug. For more information see [Using Firebug for scraping](#) and [Using Firefox for scraping](#).

After inspecting the page source, you'll find that the web sites information is inside a `<ul>` element, in fact the *second* `<ul>` element.

So we can select each `<li>` element belonging to the sites list with this code:

```
hxs.select('//ul/li')
```

And from them, the sites descriptions:

```
hxs.select('//ul/li/text()').extract()
```

The sites titles:

```
hxs.select('//ul/li/a/text()').extract()
```

And the sites links:

```
hxs.select('//ul/li/a/@href').extract()
```

As we said before, each `select()` call returns a list of selectors, so we can concatenate further `select()` calls to dig deeper into a node. We are going to use that property here, so:

```
sites = hxs.select('//ul/li')
for site in sites:
    title = site.select('a/text()').extract()
    link = site.select('a/@href').extract()
    desc = site.select('text()').extract()
    print title, link, desc
```

---

**Note:** For a more detailed description of using nested selectors, see [Nesting selectors](#) and [Working with relative XPath paths](#) in the *XPath Selectors* documentation

---

Let's add this code to our spider:

```
from scrapy.spider import BaseSpider
from scrapy.selector import HtmlXPathSelector

class DmozSpider(BaseSpider):
    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/"
]

def parse(self, response):
    hxs = HtmlXPathSelector(response)
    sites = hxs.select('//ul/li')
    for site in sites:
        title = site.select('a/text()').extract()
        link = site.select('a/@href').extract()
        desc = site.select('text()').extract()
        print title, link, desc
```

Now try crawling the dmoz.org domain again and you'll see sites being printed in your output, run:

```
scrapy crawl dmoz
```

## Using our item

`Item` objects are custom python dicts; you can access the values of their fields (attributes of the class we defined earlier) using the standard dict syntax like:

```
>>> item = DmozItem()
>>> item['title'] = 'Example title'
>>> item['title']
'Example title'
```

Spiders are expected to return their scraped data inside `Item` objects. So, in order to return the data we've scraped so far, the final code for our Spider would be like this:

```
from scrapy.spider import BaseSpider
from scrapy.selector import HtmlXPathSelector

from tutorial.items import DmozItem

class DmozSpider(BaseSpider):
    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/"
    ]

    def parse(self, response):
        hxs = HtmlXPathSelector(response)
        sites = hxs.select('//ul/li')
        items = []
        for site in sites:
            item = DmozItem()
            item['title'] = site.select('a/text()').extract()
            item['link'] = site.select('a/@href').extract()
            item['desc'] = site.select('text()').extract()
            items.append(item)
        return items
```

---

**Note:** You can find a fully-functional variant of this spider in the `dirbot` project available at

<https://github.com/scrapy/dirbot>

---

Now doing a crawl on the dmoz.org domain yields DmozItem's:

```
[dmoz] DEBUG: Scraped from <200 http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
{'desc': [u' - By David Mertz; Addison Wesley. Book in progress, full text, ASCII format. Asks
'link': [u'http://gnosis.cx/TPiP/'],
'title': [u'Text Processing in Python']}
[dmoz] DEBUG: Scraped from <200 http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>
{'desc': [u' - By Sean McGrath; Prentice Hall PTR, 2000, ISBN 0130211192, has CD-ROM. Methods to
'link': [u'http://www.informit.com/store/product.aspx?isbn=0130211192'],
'title': [u'XML Processing with Python']}
```

### 2.3.4 Storing the scraped data

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

```
scrapy crawl dmoz -o items.json -t json
```

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

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*. As with Items, 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 pipeline if you just want to store the scraped items.

### 2.3.5 Next steps

This tutorial covers only the basics of Scrapy, but there's a lot of other features not mentioned here. We recommend you continue by playing with an example project (see *Examples*), and then continue with the section *Basic concepts*.

## 2.4 Examples

The best way to learn is with examples, and Scrapy is no exception. For this reason, there is an example Scrapy project named `dirbot`, that you can use to play and learn more about Scrapy. It contains the dmoz spider described in the tutorial.

This `dirbot` project is available at: <https://github.com/scrapy/dirbot>

It contains a README file with a detailed description of the project contents.

If you're familiar with git, you can checkout the code. Otherwise you can download a tarball or zip file of the project by clicking on [Downloads](#).

There is also a site for sharing code snippets such as spiders, middlewares, extensions, or scripts, called *Scrapy snippets*. Feel free to share any code there.

*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.



---

## Basic concepts

---

### 3.1 Command line tool

New in version 0.10.

Scrapy is controlled through the `scrapy` command-line tool, to be referred here as the “Scrapy tool” to differentiate it from their 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.

#### 3.1.1 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.

Even 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
```

#### 3.1.2 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      Start crawling a spider or URL
  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, 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
```

That will create a Scrapy project under the `myproject` directory.

Next, you go inside the new project directory:

```
cd myproject
```

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 custom `USER_AGENT` per-spider setting) if the url being fetched is associated with some specific spider. This is intentional, as the `fetch` command is meant to be used to check how spiders are downloading pages.

### 3.1.3 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`
- `settings`
- `runspider`
- `shell`
- `fetch`
- `view`
- `version`

Project-only commands:

- `crawl`
- `list`
- `edit`
- `parse`
- `genspider`
- `server`
- `deploy`

### **startproject**

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

Creates a new Scrapy project named `project_name`, under the `project_name` directory.

Usage example:

```
$ scrapy startproject myproject
```

### **genspider**

- Syntax: `scrapy genspider [-t template] <name> <domain>`
- Requires project: *yes*

Create a new spider in the current project.

This is just a convenient 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.

Usage example:

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

$ scrapy genspider -d basic
from scrapy.spider import BaseSpider

class $classname(BaseSpider):
    name = "$name"
    allowed_domains = ["$domain"]
    start_urls = (
        'http://www.$domain/',
    )

    def parse(self, response):
        pass

$ scrapy genspider -t basic example example.com
Created spider 'example' using template 'basic' in module:
mybot.spiders.example
```

### crawl

- Syntax: scrapy crawl <spider>
- Requires project: *yes*

Start crawling a spider.

Usage examples:

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

### server

- Syntax: scrapy server
- Requires project: *yes*

Start Scrapyd server for this project, which can be referred from the JSON API with the project name default. For more info see: [Scrapy Service \(scrapyd\)](#).

Usage example:

```
$ scrapy server
[ ... scrapyd starts and stays idle waiting for spiders to get scheduled ... ]
```

To schedule spiders, use the Scrapyd JSON API.

### 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 convenient shortcut for the most common case, the developer is of course free to choose any tool or IDE to write and debug his spiders.

Usage example:

```
$ scrapy edit spider1
```

## fetch

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

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

The interesting thing about this command is that it fetches the page how the spider would download it. For example, if the spider has an `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 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:

```
$ 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:

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

Supported options:

- `--callback` or `-c`: spider method to use as callback for parsing the response
- `--rules` or `-r`: use [CrawlSpider](#) rules to discover the callback (ie. spider method) to use for parsing the response
- `--noitems`: don’t show extracted links
- `--nolinks`: don’t show scraped items

Usage example:

```
$ scrapy parse http://www.example.com/ -c parse_item
[ ... scrapy log lines crawling example.com spider ... ]
# Scraped Items - callback: parse -----
MyItem({'name': u'Example item',
       'category': u'Furniture',
       'length': u'12 cm'})
)
```

## 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.

## deploy

New in version 0.11.

- Syntax: `scrapy deploy [ <target:project> | -l <target> | -L ]`
- Requires project: *yes*

Deploy the project into a Scrapy server. See [Deploying your project](#).

## 3.1.4 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 custom Scrapy commands. This is used to add custom commands for your Scrapy project.

Example:

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

## 3.2 Items

The main goal in scraping is to extract structured data from unstructured sources, typically, web pages. Scrapy provides the `Item` class for this purpose.

`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.

### 3.2.1 Declaring Items

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

```
from scrapy.item import Item, Field

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

---

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

---

### 3.2.2 Item Fields

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

You can specify any kind of metadata for each field. There is no restriction on the values accepted by `Field` objects. For this same reason, there isn't a 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.

And that's all you need to know about declaring items.

### 3.2.3 Working with Items

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

#### Creating items

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

#### Getting field values

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

>>> product['price']
1000

>>> product['last_updated']
Traceback (most recent call last):
...
KeyError: 'last_updated'

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

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

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

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

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

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

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

#### Setting field values

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

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

## Accessing all populated values

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

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

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

## Other common tasks

Copying items:

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

Creating dicts from items:

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

Creating items from dicts:

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

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

## 3.2.4 Extending Items

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

For example:

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

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

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



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

### 3.2.5 Item objects

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

Return a new Item optionally initialized from the given argument.

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

**fields**

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

### 3.2.6 Field objects

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

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

## 3.3 Spiders

Spiders are classes which define how a certain site (or domain) will be scraped, including how to crawl the site and how to extract scraped items from their pages. In other words, Spiders are the place where you define the custom behaviour for crawling and parsing pages for a particular site.

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 `Item` objects, `Request` objects, or an iterable of both. 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 *XPath 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 in some Item pipeline.

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

### 3.3.1 Built-in spiders reference

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 a 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:

```
from scrapy.item import Item

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

## BaseSpider

### `class scrapy.spider.BaseSpider`

This is the simplest spider, and the one from which every other spider must inherit from (either the ones that come bundled with Scrapy, or the ones that you write yourself). It doesn't provide any special functionality. It just requests the given `start_urls/start_requests`, 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.

Is recommended to name your spiders after the domain that their crawl.

#### **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 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.

#### **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:

```
def start_requests(self):
    return [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 `BaseSpider` class.

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

**Parameters** `response` – the response to parse

**log (message[, level, component])**

Log a message using the `scrapy.log.msg()` function, automatically populating the spider argument with the `name` of this spider. For more information see *Logging*.

**BaseSpider example**

Let's see an example:

```
from scrapy import log # This module is useful for printing out debug information
from scrapy.spider import BaseSpider

class MySpider(BaseSpider):
    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.log('A response from %s just arrived!' % response.url)
```

Another example returning multiples Requests and Items from a single callback:

```
from scrapy.selector import HtmlXPathSelector
from scrapy.spider import BaseSpider
from scrapy.http import Request
from myproject.items import MyItem

class MySpider(BaseSpider):
    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):
```

```
hxs = HtmlXPathSelector(response)
for h3 in hxs.select('//h3').extract():
    yield MyItem(title=h3)

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

## CrawlSpider

**class** scrapy.contrib.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 BaseSpider (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.

## Crawling rules

**class** scrapy.contrib.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 default 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:

```

from scrapy.contrib.spiders import CrawlSpider, Rule
from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor
from scrapy.selector import HtmlXPathSelector
from scrapy.item import Item

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(SgmlLinkExtractor(allow=('category\.php', ), deny=('subsection\.php', ))),

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

    def parse_item(self, response):
        self.log('Hi, this is an item page! %s' % response.url)

        hxs = HtmlXPathSelector(response)
        item = Item()
        item['id'] = hxs.select('//td[@id="item_id"]/text()').re(r'ID: (\d+)')
        item['name'] = hxs.select('//td[@id="item_name"]/text()').extract()
        item['description'] = hxs.select('//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 a `Item` will be filled with it.

## XMLFeedSpider

### class scrapy.contrib.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 `HtmlXPathSelector`. 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 `XmlXPathSelector`. Keep in mind this uses DOM parsing and must load all DOM in memory which could be a problem for big feeds

It defaults to: `'iternodes'`.

#### itertag

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

```
itertag = 'product'
```

#### namespaces

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

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

Example:

```
class YourSpider(XMLFeedSpider):

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

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

#### `adapt_response(response)`

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

#### `parse_node(response, selector)`

This method is called for the nodes matching the provided tag name (`itertag`). Receives the response and an XPathSelector 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:

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

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

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

        item = Item()
        item['id'] = node.select('@id').extract()
        item['name'] = node.select('name').extract()
```

```

item['description'] = node.select('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.contrib.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).

#### **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`:

```

from scrapy import log
from scrapy.contrib.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 = ';'
    headers = ['id', 'name', 'description']

    def parse_row(self, response, row):
        log.msg('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.contrib.spiders.SitemapSpider`

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

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

#### **sitemap\_urls**

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

You can also point to a [robots.txt](#) and it will be parsed to extract sitemap urls from it.

#### **sitemap\_rules**

A list of tuples (regex, callback) where:

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

For example:

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

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

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

#### **sitemap\_follow**

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

By default, all sitemaps are followed.

### **SitemapSpider examples**

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

```
from scrapy.contrib.spiders import SitemapSpider

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

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

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

```
from scrapy.contrib.spiders import SitemapSpider

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

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

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

Follow sitemaps defined in the [robots.txt](#) file and only follow sitemaps whose url contains /sitemap\_shop:

```
from scrapy.contrib.spiders import SitemapSpider

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



```
sitemap_rules = [
    ('/shop/', 'parse_shop'),
]
sitemap_follow = ['/sitemap_shops']

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

Combine SitemapSpider with other sources of urls:

```
from scrapy.contrib.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 += [Request(x, callback=self.parse_other) for x in self.other_urls]
        return requests

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

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

## 3.4 Link Extractors

LinkExtractors are objects whose only purpose is to extract links from web pages (`scrapy.http.Response` objects) which will be eventually followed.

There are two Link Extractors available in Scrapy by default, but you create your own custom Link Extractors to suit your needs by implementing a simple interface.

The only public method that every LinkExtractor has is `extract_links`, which receives a `Response` object and returns a list of links. Link Extractors are meant to be instantiated once and their `extract_links` method called several times with different responses, to extract links to follow.

Link extractors are used in the `CrawlSpider` class (available in Scrapy), through a set of rules, but you can also use it in your spiders, even if you don't subclass from `CrawlSpider`, as its purpose is very simple: to extract links.

### 3.4.1 Built-in link extractors reference

All available link extractors classes bundled with Scrapy are provided in the `scrapy.contrib.linkextractors` module.

## SgmlLinkExtractor

```
class scrapy.contrib.linkextractors.sgml.SgmlLinkExtractor (allow=(), deny=(),
                                                            allow_domains=(),
                                                            deny_domains=(),
                                                            deny_extensions=None,
                                                            restrict_xpaths=(),
                                                            tags=('a', 'area'),
                                                            attrs=('href'), canonical-
                                                            ize=True, unique=True,
                                                            process_value=None)
```

The SgmlLinkExtractor extends the base `BaseSgmlLinkExtractor` by providing additional filters that you can specify to extract links, including regular expressions patterns that the links must match to be extracted. All those filters are configured through these constructor parameters:

### 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 list of extensions that should be ignored when extracting links. If not given, it will default to the `IGNORED_EXTENSIONS` list defined in the `scrapy.linkextractor` module.
- **restrict\_xpaths** (*str or list*) – is a 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.
- **tags** (*str or list*) – a tag or a list of tags to consider when extracting links. Defaults to `('a', 'area')`.
- **attrs** (*boolean*) – 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 `scrapy.utils.url.canonicalize_url`). Defaults to `True`.
- **unique** (*boolean*) – whether duplicate filtering should be applied to extracted links.
- **process\_value** (*callable*) – see `process_value` argument of `BaseSgmlLinkExtractor` class constructor

## BaseSgmlLinkExtractor

```
class scrapy.contrib.linkextractors.sgml.BaseSgmlLinkExtractor(tag="a",
                                                              attr="href",
                                                              unique=False, process_value=None)
```

The purpose of this Link Extractor is only to serve as a base class for the `SgmlLinkExtractor`. You should use that one instead.

The constructor arguments are:

### Parameters

- **tag** (*str or callable*) – either a string (with the name of a tag) or a function that receives a tag name and returns `True` if links should be extracted from that tag, or `False` if they shouldn't. Defaults to `'a'`. request (once it's downloaded) as its first parameter. For more information, see *Passing additional data to callback functions*.
- **attr** (*str or callable*) – either string (with the name of a tag attribute), or a function that receives an attribute name and returns `True` if links should be extracted from it, or `False` if they shouldn't. Defaults to `href`.
- **unique** (*boolean*) – is a boolean that specifies if a duplicate filtering should be applied to links extracted.
- **process\_value** (*callable*) – a function which receives each value extracted from the tag and attributes scanned and can modify the value and return a new one, or return `None` to ignore the link altogether. If not given, `process_value` defaults to `lambda x: x`.

For example, to extract links from this code:

```
<a href="javascript:goToPage('../other/page.html'); return false">Link text</a>
```

You can use the following function in `process_value`:

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

## 3.5 XPath 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 screen 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 a XML parsing library (which also parses HTML) with a pythonic API based on `ElementTree` (which is not part of the Python standard library).

Scrapy comes with its own mechanism for extracting data. They're called XPath selectors (or just "selectors", for short) because they "select" certain parts of the HTML document specified by `XPath` expressions.

`XPath` is a language for selecting nodes in XML documents, which can also be used with HTML.

Both `lxml` and Scrapy Selectors are built over the `libxml2` 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 the *XPath selector reference*.

### 3.5.1 Using selectors

#### Constructing selectors

There are two types of selectors bundled with Scrapy. Those are:

- `HtmlXPathSelector` - for working with HTML documents
- `XmlXPathSelector` - for working with XML documents

Both share the same selector API, and are constructed with a `Response` object as their first parameter. This is the `Response` they're going to be “selecting”.

Example:

```
hxs = HtmlXPathSelector(response) # a HTML selector
xbs = XmlXPathSelector(response) # a XML selector
```

#### Using selectors with XPaths

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:

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

Here's its HTML code:

```
<html>
<head>
  <base href='http://example.com/' />
  <title>Example website</title>
</head>
<body>
  <div id='images'>
    <a href='image1.html'>Name: My image 1 <br /><img src='image1_thumb.jpg' /></a>
    <a href='image2.html'>Name: My image 2 <br /><img src='image2_thumb.jpg' /></a>
    <a href='image3.html'>Name: My image 3 <br /><img src='image3_thumb.jpg' /></a>
    <a href='image4.html'>Name: My image 4 <br /><img src='image4_thumb.jpg' /></a>
    <a href='image5.html'>Name: My image 5 <br /><img src='image5_thumb.jpg' /></a>
  </div>
</body>
</html>
```

First, let's open the shell:

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

Then, after the shell loads, you'll have some selectors already instantiated and ready to use.

Since we're dealing with HTML, we'll be using the `HtmlXPathSelector` object which is found, by default, in the `hxs` shell variable.

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

```
>>> hxs.select('//title/text()')
[<HtmlXPathSelector (text) xpath=//title/text()>]
```

As you can see, the `select()` method returns an `XPathSelectorList`, which is a list of new selectors. This API can be used quickly for extracting nested data.

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

```
>>> hxs.select('//title/text()').extract()
[u'Example website']
```

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

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

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

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

## Using selectors with regular expressions

Selectors also have a `re()` method for extracting data using regular expressions. However, unlike using the `select()` method, the `re()` method does not return a list of `XPathSelector` objects, so you can't construct nested `.re()` calls.

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

```
>>> hxs.select('//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']
```

## Nesting selectors

The `select()` selector method returns a list of selectors, so you can call the `select()` for those selectors too. Here's an example:

```
>>> links = hxs.select('//a[contains(@href, "image")]')
>>> links.extract()
[u'<a href="image1.html">Name: My image 1 <br></a>',
 u'<a href="image2.html">Name: My image 2 <br></a>',
```

```
u'<a href="image3.html">Name: My image 3 <br></a>',
u'<a href="image4.html">Name: My image 4 <br></a>',
u'<a href="image5.html">Name: My image 5 <br></a>']
```

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

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

### Working with relative XPath

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

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

```
>>> divs = hxs.select('//div')
```

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

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

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

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

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

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

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

### 3.5.2 Built-in XPath Selectors reference

There are two types of selectors bundled with Scrapy: `HtmlXPathSelector` and `XmlXPathSelector`. Both of them implement the same `XPathSelector` interface. The only different is that one is used to process HTML data and the other XML data.

#### XPathSelector objects

**class scrapy.selector.XPathSelector(response)**

A `XPathSelector` object is a wrapper over response to select certain parts of its content.

response is a `Response` object that will be used for selecting and extracting data

**select** (*xpath*)

Apply the given XPath relative to this `XPathSelector` and return a list of `XPathSelector` objects (ie. a `XPathSelectorList`) with the result.

*xpath* is a string containing the XPath to apply

**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)`

**extract** ()

Return a unicode string with the content of this `XPathSelector` object.

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

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

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

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

**XPathSelectorList objects****class scrapy.selector.XPathSelectorList**

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

**select** (*xpath*)

Call the `XPathSelector.select()` method for all `XPathSelector` objects in this list and return their results flattened, as a new `XPathSelectorList`.

*xpath* is the same argument as the one in `XPathSelector.select()`

**re** (*regex*)

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

*regex* is the same argument as the one in `XPathSelector.re()`

**extract** ()

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

**extract\_unquoted** ()

Call the `XPathSelector.extract_unquoted()` method for all `XPathSelector` objects in this list and return their results flattened, as a list of unicode strings. This method should not be applied to all kinds of `XPathSelectors`. For more info see `XPathSelector.extract_unquoted()`.

**HtmlXPathSelector objects****class scrapy.selector.HtmlXPathSelector** (*response*)

A subclass of `XPathSelector` for working with HTML content. It uses the `libxml2` HTML parser. See the `XPathSelector` API for more info.

### HtmlXPathSelector examples

Here's a couple of `HtmlXPathSelector` examples to illustrate several concepts. In all cases, we assume there is already an `HtmlPathSelector` instantiated with a `Response` object like this:

```
x = HtmlXPathSelector(html_response)
```

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

```
x.select("//h1")
```

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

```
x.select("//h1").extract()           # this includes the h1 tag
x.select("//h1/text()").extract()    # this excludes the h1 tag
```

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

```
for node in x.select("//p"):
...     print node.select("@href")
```

4. Extract textual data from all `<p>` tags without entities, as a list of unicode strings:

```
x.select("//p/text()").extract_unquoted()

# the following line is wrong. extract_unquoted() should only be used
# with textual XPathSelectors
x.select("//p").extract_unquoted()  # it may work but output is unpredictable
```

### XmlXPathSelector objects

**class** scrapy.selector.**XmlXPathSelector**(response)

A subclass of `XPathSelector` for working with XML content. It uses the `libxml2` XML parser. See the `XPathSelector` API for more info.

### XmlXPathSelector examples

Here's a couple of `XmlXPathSelector` examples to illustrate several concepts. In all cases we assume there is already a `XmlPathSelector` instantiated with a `Response` object like this:

```
x = HtmlXPathSelector(xml_response)
```

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

```
x.select("//h1")
```

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

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



## 3.6 Item Loaders

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

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

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

### 3.6.1 Using Item Loaders to populate items

To use an Item Loader, you must first instantiate it. You can either instantiate it with an 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 *XPath Selectors*. You can add more than one value to the same item field; the Item Loader will know how to “join” those values later using a proper processing function.

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

```
from scrapy.contrib.loader import XPathItemLoader
from myproject.items import Product

def parse(self, response):
    l = XPathItemLoader(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_xpath('stock', '//p[@id="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, 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 populates and returns the item populated with the data previously extracted and collected with the `add_xpath()` and `add_value()` calls.

### 3.6.2 Input and Output processors

An Item Loader contains one input processor and one output processor for each (item) field. The input processor processes the extracted data as soon as it's received (through the `add_xpath()` or `add_value()` methods) and the result of the input processor is collected and kept inside the ItemLoader. After collecting all data, the

`ItemLoader.load_item()` method is called to populate and get the populated `Item` object. That's when the output processor is called with the data previously collected (and processed using the input processor). The result of the output processor is the final value that gets assigned to the item.

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

```
l = XPathItemLoader(Product(), some_xpath_selector)
l.add_xpath('name', xpath1) # (1)
l.add_xpath('name', xpath2) # (2)
l.add_value('name', 'test') # (3)
return l.load_item() # (4)
```

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 value to be collected is assigned directly, instead of being extracted from a XPath. 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.
4. The data collected in (1) and (2) 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.

### 3.6.3 Declaring Item Loaders

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

```
from scrapy.contrib.loader import ItemLoader
from scrapy.contrib.loader.processor import TakeFirst, MapCompose, Join

class ProductLoader(ItemLoader):

    default_output_processor = TakeFirst()

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

    price_in = MapCompose(unicode.strip)

    # ...
```

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

### 3.6.4 Declaring Input and Output Processors

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

```
from scrapy.item import Item, Field
from scrapy.contrib.loader.processor import MapCompose, Join, TakeFirst

from scrapy.utils.markup import remove_entities
from myproject.utils import filter_prices

class Product(Item):
    name = Field(
        input_processor=MapCompose(remove_entities),
        output_processor=Join(),
    )
    price = Field(
        default=0,
        input_processor=MapCompose(remove_entities, filter_prices),
        output_processor=TakeFirst(),
    )
```

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

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

See also: *Reusing and extending Item Loaders*.

### 3.6.5 Item Loader Context

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

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

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

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

There are several ways to modify Item Loader context values:

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

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

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

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

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

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

### 3.6.6 ItemLoader objects

**class** scrapy.contrib.loader.ItemLoader([item], \*\*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`.

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

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

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

Available keyword arguments:

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

Examples:

```
>>> from scrapy.contrib.loader.processor 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 appened to the data collected for that field. If the field already contains collected data, the new data is added.

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

Examples:

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

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

Similar to `add_value()` but replaces the collected data with the new value 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.

**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.

**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.

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

The *XPathItemLoader* class extends the *ItemLoader* class providing more convenient mechanisms for extracting data from web pages using *XPath selectors*.

*XPathItemLoader* objects accept two more additional parameters in their constructors:

#### Parameters

- **selector** (*XPathSelector* object) – The selector to extract data from, when using the *add\_xpath()* or *replace\_xpath()* 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.

**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 *XPathItemLoader*.

#### Parameters

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

Examples:

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

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

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

See `get_xpath()` for *kwargs*.

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

Examples:

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

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

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

**default\_selector\_class**

The class used to construct the `selector` of this `XPathItemLoader`, 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 `XPathSelector` object to extract data from. It's either the selector given in the constructor or one created from the response given in the constructor using the `default_selector_class`. This attribute is meant to be read-only.

### 3.6.7 Reusing and extending Item Loaders

As your project grows bigger and acquires more and more spiders, maintenance becomes a fundamental problem, specially 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 (ie. ---Plasma TV---) and you don't want to end up scraping those dashes in the final product names.

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

```
from scrapy.contrib.loader.processor import MapCompose
from myproject.ItemLoaders import ProductLoader

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

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

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

```
from scrapy.contrib.loader.processor import MapCompose
from myproject.ItemLoaders import ProductLoader
```

```
from myproject.utils.xml import remove_cdata

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

And that's how you typically extend input processors.

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

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

### 3.6.8 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.contrib.loader.processor.Identity**

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

Example:

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

**class scrapy.contrib.loader.processor.TakeFirst**

Return 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 accept Loader contexts.

Example:

```
>>> from scrapy.contrib.loader.processor import TakeFirst
>>> proc = TakeFirst()
>>> proc(['', 'one', 'two', 'three'])
'one'
```

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

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

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

Example:

```
>>> from scrapy.contrib.loader.processor 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.contrib.loader.processor.**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 each element is passed to the first function, and the result of that function (for each element) is concatenated to construct a new iterable, which is then passed to the second function, and so on, until the last function is applied for each value of the list of values collected so far. The output values of the last function are concatenated together to produce the output of this processor.

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

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

The example below should clarify how it works:

```
>>> def filter_world(x):
...     return None if x == 'world' else x
...
>>> from scrapy.contrib.loader.processor 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.

## 3.7 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 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 XPaths 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.

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.

### 3.7.1 Launch the shell

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

```
scrapy shell <url>
```

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

### 3.7.2 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 [XPathSelector](#) objects (for both HTML and XML content).

Those objects are:

- `spider` - the Spider which is known to handle the URL, or a [BaseSpider](#) 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
- `hxs` - a [HtmlXPathSelector](#) object constructed with the last response fetched
- `xxs` - a [XmlXPathSelector](#) object constructed with the last response fetched
- `settings` - the current *Scrapy settings*

### 3.7.3 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 <http://slashdot.org> page. Finally, we modify the (Slashdot) request method to POST and re-fetch it getting a HTTP 405 (method not allowed) 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 objects
[s]   hxs          <HtmlXPathSelector (http://scrapy.org) xpath=None>
[s]   item         Item()
[s]   request      <http://scrapy.org>
[s]   response     <http://scrapy.org>
[s]   settings     <Settings 'mybot.settings'>
[s]   spider       <scrapy.spider.models.BaseSpider object at 0x2bed9d0>
[s]   xxs          <XmlXPathSelector (http://scrapy.org) xpath=None>
[s] Useful shortcuts:
[s]   shelp()       Prints this help.
[s]   fetch(req_or_url) Fetch a new request or URL and update objects
[s]   view(response) View response in a browser

>>>
```

After that, we can start playing with the objects:

```
>>> hxs.select("//h2/text()").extract()[0]
u'Welcome to Scrapy'

>>> fetch("http://slashdot.org")
[s] Available Scrapy objects:
[s]   hxs          <HtmlXPathSelector (http://slashdot.org) xpath=None>
[s]   item         JobItem()
[s]   request      <GET http://slashdot.org>
[s]   response     <200 http://slashdot.org>
[s]   settings     <Settings 'jobshot.settings'>
[s]   spider       <BaseSpider 'default' at 0x3c44a10>
[s]   xxs          <XmlXPathSelector (http://slashdot.org) xpath=None>
[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

>>> hxs.select("//h2/text()").extract()
[u'News for nerds, stuff that matters']

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

>>> fetch(request)
2009-04-03 00:57:39-0300 [default] ERROR: Downloading <http://slashdot.org> from <None>: 405 Method Not Allowed

>>>
```

### 3.7.4 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:

```
class MySpider(BaseSpider):
    ...

    def parse(self, response):
        if response.url == 'http://www.example.com/products.php':
            from scrapy.shell import inspect_response
            inspect_response(response)

        # ... your parsing code ..
```

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

```
2009-08-27 19:15:25-0300 [example.com] DEBUG: Crawled <http://www.example.com/> (referer: <None>)
2009-08-27 19:15:26-0300 [example.com] DEBUG: Crawled <http://www.example.com/products.php> (referer: <None>)
[s] Available objects
[s]   hxs          <HtmlXPathSelector (http://www.example.com/products.php) xpath=None>
...

>>> response.url
'http://www.example.com/products.php'
```

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

```
>>> hxs.select('//h1')
[]
```

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

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

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

```
>>> ^D
2009-08-27 19:15:25-0300 [example.com] DEBUG: Crawled <http://www.example.com/product.php?id=1> (referer: <None>)
2009-08-27 19:15:25-0300 [example.com] DEBUG: Crawled <http://www.example.com/product.php?id=2> (referer: <None>)
# ...
```

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

## 3.8 Item Pipeline

After an item has been scraped by a spider, it is sent to the Item Pipeline which process 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 use for item pipelines are:

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

### 3.8.1 Writing your own item pipeline

Writing your own item pipeline is easy. Each item pipeline component is a single Python class that must implement the following method:

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

This method is called for every item pipeline component and must either return a `Item` (or any descendant class) object or raise a `DropItem` exception. Dropped items are no longer processed by further pipeline components.

**Parameters**

- **item** (`Item` object) – the item scraped
- **spider** (`BaseSpider` object) – the spider which scraped the item

Additionally, they may also implement the following methods:

**open\_spider**(*spider*)

This method is called when the spider is opened.

**Parameters** **spider** (`BaseSpider` object) – the spider which was opened

**close\_spider**(*spider*)

This method is called when the spider is closed.

**Parameters** **spider** (`BaseSpider` object) – the spider which was closed

### 3.8.2 Item pipeline example

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

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

```
from scrapy.exceptions import DropItem

class PricePipeline(object):

    vat_factor = 1.15

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

## Write items to a JSON file

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

```
import json

class JsonWriterPipeline(object):

    def __init__(self):
        self.file = open('items.jsonl', 'wb')

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

---

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

---

### 3.8.3 Activating an Item Pipeline component

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

```
ITEM_PIPELINES = [
    'myproject.pipeline.PricePipeline',
    'myproject.pipeline.JsonWriterPipeline',
]
```

### 3.8.4 Item pipeline example with resources per spider

Sometimes you need to keep resources about the items processed grouped per spider, and delete those resource when a spider finishes.

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

```
from scrapy.xlib.pydispatch import dispatcher
from scrapy import signals
from scrapy.exceptions import DropItem

class DuplicatesPipeline(object):

    def __init__(self):
        self.duplicates = {}
        dispatcher.connect(self.spider_opened, signals.spider_opened)
        dispatcher.connect(self.spider_closed, signals.spider_closed)

    def spider_opened(self, spider):
        self.duplicates[spider] = set()

    def spider_closed(self, spider):
        del self.duplicates[spider]

    def process_item(self, item, spider):
```

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

## 3.9 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 a “export file” with the scraped data (commonly called “export feed”) to be consumed by other systems.

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

### 3.9.1 Serialization formats

For serializing the scraped data, the feed exports use the *Item exporters* and 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

- `FEED_FORMAT`: `json`
- Exporter used: `JsonItemExporter`
- See *this warning* if you’re using JSON with large feeds

#### JSON lines

- `FEED_FORMAT`: `jsonlines`
- Exporter used: `JsonLinesItemExporter`

#### CSV

- `FEED_FORMAT`: `csv`
- Exporter used: `CsvItemExporter`

## XML

- `FEED_FORMAT`: `xml`
- Exporter used: `XmlItemExporter`

## Pickle

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

## Marshal

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

## 3.9.2 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 `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 `boto` library is installed.

## 3.9.3 Storage URI parameters

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

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

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

Here are some examples to illustrate:

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

## 3.9.4 Storage backends

### Local filesystem

The feeds are stored in the local filesystem.

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

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

### FTP

The feeds are stored in a FTP server.

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

### S3

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

- URI scheme: `s3`
- Example URIs:
  - `s3://mybucket/path/to/export.csv`
  - `s3://aws_key:aws_secret@mybucket/path/to/export.csv`
- Required external libraries: `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`



### 3.9.5 Settings

These are the settings used for configuring the feed exports:

- `FEED_URI` (mandatory)
- `FEED_FORMAT`
- `FEED_STORAGES`
- `FEED_EXPORTERS`
- `FEED_STORE_EMPTY`

#### **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\_STORE\_EMPTY**

Default: `False`

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

#### **FEED\_STORAGES**

Default:: `{}`

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

#### **FEED\_STORAGES\_BASE**

Default:

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

A dict containing the built-in feed storage backends supported by Scrapy.

## FEED\_EXPORTERS

Default:: {}

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

## FEED\_EXPORTERS\_BASE

Default:

```
FEED_EXPORTERS_BASE = {
    'json': 'scrapy.contrib.exporter.JsonItemExporter',
    'jsonlines': 'scrapy.contrib.exporter.JsonLinesItemExporter',
    'csv': 'scrapy.contrib.exporter.CsvItemExporter',
    'xml': 'scrapy.contrib.exporter.XmlItemExporter',
    'marshal': 'scrapy.contrib.exporter.MarshalItemExporter',
}
```

A dict containing the built-in feed exporters supported by Scrapy.

## 3.10 Link Extractors

LinkExtractors are objects whose only purpose is to extract links from web pages (`scrapy.http.Response` objects) which will be eventually followed.

There are two Link Extractors available in Scrapy by default, but you create your own custom Link Extractors to suit your needs by implementing a simple interface.

The only public method that every LinkExtractor has is `extract_links`, which receives a `Response` object and returns a list of links. Link Extractors are meant to be instantiated once and their `extract_links` method called several times with different responses, to extract links to follow.

Link extractors are used in the `CrawlSpider` class (available in Scrapy), through a set of rules, but you can also use it in your spiders, even if you don't subclass from `CrawlSpider`, as its purpose is very simple: to extract links.

### 3.10.1 Built-in link extractors reference

All available link extractors classes bundled with Scrapy are provided in the `scrapy.contrib.linkextractors` module.

#### SgmlLinkExtractor

```
class scrapy.contrib.linkextractors.sgml.SgmlLinkExtractor(allow=(), deny=(),
    allow_domains=(),
    deny_domains=(),
    deny_extensions=None,
    restrict_xpaths=(),
    tags=('a', 'area'),
    attrs=('href'), canonicalize=True, unique=True,
    process_value=None)
```

The `SgmlLinkExtractor` extends the base `BaseSgmlLinkExtractor` by providing additional filters that you

can specify to extract links, including regular expressions patterns that the links must match to be extracted. All those filters are configured through these constructor parameters:

#### 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 list of extensions that should be ignored when extracting links. If not given, it will default to the `IGNORED_EXTENSIONS` list defined in the `scrapy.linkextractor` module.
- **restrict\_xpaths** (*str or list*) – is a 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.
- **tags** (*str or list*) – a tag or a list of tags to consider when extracting links. Defaults to `('a', 'area')`.
- **attrs** (*boolean*) – 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 `scrapy.utils.url.canonicalize_url`). Defaults to `True`.
- **unique** (*boolean*) – whether duplicate filtering should be applied to extracted links.
- **process\_value** (*callable*) – see `process_value` argument of `BaseSgmlLinkExtractor` class constructor

### BaseSgmlLinkExtractor

```
class scrapy.contrib.linkextractors.sgml.BaseSgmlLinkExtractor(tag="a",
                                                                attr="href",
                                                                unique=False, process_value=None)
```

The purpose of this Link Extractor is only to serve as a base class for the `SgmlLinkExtractor`. You should use that one instead.

The constructor arguments are:

#### Parameters

- **tag** (*str or callable*) – either a string (with the name of a tag) or a function that receives a tag name and returns `True` if links should be extracted from that tag, or `False` if they shouldn't. Defaults to `'a'`. request (once it's downloaded) as its first parameter. For more information, see *Passing additional data to callback functions*.

- **attr** (*str or callable*) – either string (with the name of a tag attribute), or a function that receives an attribute name and returns `True` if links should be extracted from it, or `False` if they shouldn't. Defaults to `href`.
- **unique** (*boolean*) – is a boolean that specifies if a duplicate filtering should be applied to links extracted.
- **process\_value** (*callable*) – a function which receives each value extracted from the tag and attributes scanned and can modify the value and return a new one, or return `None` to ignore the link altogether. If not given, `process_value` defaults to `lambda x: x`.

For example, to extract links from this code:

```
<a href="javascript:goToPage('../other/page.html'); return false">Link text</a>
```

You can use the following function in `process_value`:

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

**Command line tool** Learn about the command-line tool used to manage your Scrapy project.

**Items** Define the data you want to scrape.

**Spiders** Write the rules to crawl your websites.

**XPath Selectors** Extract the data from web pages.

**Scrapy shell** Test your extraction code in an interactive environment.

**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.

**Link Extractors** Convenient classes to extract links to follow from pages.

---

## Built-in services

---

### 4.1 Logging

Scrapy provides a logging facility which can be used through the `scrapy.log` module. The current underlying implementation uses `Twisted logging` but this may change in the future.

The logging service must be explicitly started through the `scrapy.log.start()` function.

#### 4.1.1 Log levels

Scrapy provides 5 logging levels:

1. `CRITICAL` - for critical errors
2. `ERROR` - for regular errors
3. `WARNING` - for warning messages
4. `INFO` - for informational messages
5. `DEBUG` - for debugging messages

#### 4.1.2 How to set the log level

You can set the log level using the `-loglevel/-L` command line option, or using the `LOG_LEVEL` setting.

#### 4.1.3 How to log messages

Here's a quick example of how to log a message using the `WARNING` level:

```
from scrapy import log
log.msg("This is a warning", level=log.WARNING)
```

#### 4.1.4 Logging from Spiders

The recommended way to log from spiders is by using the `Spider log()` method, which already populates the `spider` argument of the `scrapy.log.msg()` function. The other arguments are passed directly to the `msg()` function.

### 4.1.5 scrapy.log module

`scrapy.log.started`

A boolean which is `True` if logging has been started or `False` otherwise.

`scrapy.log.start` (*logfile=None, loglevel=None, logstdout=None*)

Start the logging facility. This must be called before actually logging any messages. Otherwise, messages logged before this call will get lost.

#### Parameters

- **logfile** (*str*) – the file path to use for logging output. If omitted, the `LOG_FILE` setting will be used. If both are `None`, the log will be sent to standard error.
- **loglevel** – the minimum logging level to log. Available values are: `CRITICAL`, `ERROR`, `WARNING`, `INFO` and `DEBUG`.
- **logstdout** (*boolean*) – if `True`, all standard output (and error) of your application will be logged instead. For example if you “print ‘hello’” it will appear in the Scrapy log. If omitted, the `LOG_STDOUT` setting will be used.

`scrapy.log.msg` (*message, level=INFO, spider=None*)

Log a message

#### Parameters

- **message** (*str*) – the message to log
- **level** – the log level for this message. See [Log levels](#).
- **spider** (`BaseSpider` object) – the spider to use for logging this message. This parameter should always be used when logging things related to a particular spider.

`scrapy.log.CRITICAL`

Log level for critical errors

`scrapy.log.ERROR`

Log level for errors

`scrapy.log.WARNING`

Log level for warnings

`scrapy.log.INFO`

Log level for informational messages (recommended level for production deployments)

`scrapy.log.DEBUG`

Log level for debugging messages (recommended level for development)

### 4.1.6 Logging settings

These settings can be used to configure the logging:

- `LOG_ENABLED`
- `LOG_ENCODING`
- `LOG_FILE`
- `LOG_LEVEL`
- `LOG_STDOUT`

## 4.2 Stats Collection

### 4.2.1 Overview

Scrapy provides a convenient service for collecting stats in the form of key/values, both globally and per spider. It's called the Stats Collector, and it's a singleton which can be imported and used quickly, as illustrated by the examples in the *Common Stats Collector uses* section below.

The stats collection is enabled by default but can be disabled through the `STATS_ENABLED` setting.

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 one stats table per open spider and one global stats table. You can't set or get stats from a closed spider, but the spider-specific stats table is automatically opened when the spider is opened, and closed when the spider is closed.

### 4.2.2 Common Stats Collector uses

Import the stats collector:

```
from scrapy.stats import stats
```

Set global stat value:

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

Increment global stat value:

```
stats.inc_value('spiders_crawled')
```

Set global stat value only if greater than previous:

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

Set global stat value only if lower than previous:

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

Get global stat value:

```
>>> stats.get_value('spiders_crawled')
8
```

Get all global stats (ie. not particular to any spider):

```
>>> stats.get_stats()
{'hostname': 'localhost', 'spiders_crawled': 8}
```

Set spider specific stat value (spider stats must be opened first, but this task is handled automatically by the Scrapy engine):

```
stats.set_value('start_time', datetime.now(), spider=some_spider)
```

Where `some_spider` is a `BaseSpider` object.

Increment spider-specific stat value:

```
stats.inc_value('pages_crawled', spider=some_spider)
```

Set spider-specific stat value only if greater than previous:

```
stats.max_value('max_items_scraped', value, spider=some_spider)
```

Set spider-specific stat value only if lower than previous:

```
stats.min_value('min_free_memory_percent', value, spider=some_spider)
```

Get spider-specific stat value:

```
>>> stats.get_value('pages_crawled', spider=some_spider)
1238
```

Get all stats from a given spider:

```
>>> stats.get_stats(spider=some_spider)
{'pages_crawled': 1238, 'start_time': datetime.datetime(2009, 7, 14, 21, 47, 28, 977139)}
```

## 4.2.3 Stats Collector API

There are several Stats Collectors available under the `scrapy.statscol` module and they all implement the Stats Collector API defined by the `StatsCollector` class (which they all inherit from).

```
class scrapy.statscol.StatsCollector
```

**get\_value** (*key*, *default=None*, *spider=None*)

Return the value for the given stats key or default if it doesn't exist. If `spider` is `None` the global stats table is consulted, otherwise the spider specific one is. If the spider is not yet opened a `KeyError` exception is raised.

**get\_stats** (*spider=None*)

Get all stats from the given spider (if `spider` is given) or all global stats otherwise, as a dict. If `spider` is not opened `KeyError` is raised.

**set\_value** (*key*, *value*, *spider=None*)

Set the given value for the given stats key on the global stats (if `spider` is not given) or the spider-specific stats (if `spider` is given), which must be opened or a `KeyError` will be raised.

**set\_stats** (*stats*, *spider=None*)

Set the given stats (as a dict) for the given spider. If the spider is not opened a `KeyError` will be raised.

**inc\_value** (*key*, *count=1*, *start=0*, *spider=None*)

Increment the value of the given stats key, by the given count, assuming the start value given (when it's not set). If `spider` is not given the global stats table is used, otherwise the spider-specific stats table is used, which must be opened or a `KeyError` will be raised.

**max\_value** (*key*, *value*, *spider=None*)

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. If `spider` is not given, the global stats table is used, otherwise the spider-specific stats table is used, which must be opened or a `KeyError` will be raised.



**min\_value** (*key, value, spider=None*)

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. If spider is not given, the global stats table is used, otherwise the spider-specific stats table is used, which must be opened or a `KeyError` will be raised.

**clear\_stats** (*spider=None*)

Clear all global stats (if spider is not given) or all spider-specific stats if spider is given, in which case it must be opened or a `KeyError` will be raised.

**iter\_spider\_stats** ()

Return a iterator over (*spider, spider\_stats*) for each open spider currently tracked by the stats collector, where *spider\_stats* is the dict containing all spider-specific stats.

Global stats are not included in the iterator. If you want to get those, use `get_stats()` method.

**open\_spider** (*spider*)

Open the given spider for stats collection. This method must be called prior to working with any stats specific to that spider, but this task is handled automatically by the Scrapy engine.

**close\_spider** (*spider*)

Close the given spider. After this is called, no more specific stats for this spider can be accessed. This method is called automatically on the `spider_closed` signal.

**engine\_stopped** ()

Called after the engine is stopped, to dump or persist global stats.

## 4.2.4 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`.

When stats are disabled (through the `STATS_ENABLED` setting) the `STATS_CLASS` setting is ignored and the `DummyStatsCollector` is used.

### MemoryStatsCollector

**class** scrapy.statscol.**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.statscol.**DummyStatsCollector**

A Stats collector which does nothing but is very efficient. This is the Stats Collector used when stats are disabled (through the `STATS_ENABLED` setting).

## 4.2.5 Stats signals

The Stats Collector provides some signals for extending the stats collection functionality:

`scrapy.signals.stats_spider_opened(spider)`

Sent right after the stats spider is opened. You can use this signal to add startup stats for the spider (example: start time).

**Parameters** `spider` (*str*) – the stats spider just opened

`scrapy.signals.stats_spider_closing(spider, reason)`

Sent just before the stats spider is closed. You can use this signal to add some closing stats (example: finish time).

**Parameters**

- **spider** (*str*) – the stats spider about to be closed
- **reason** (*str*) – the reason why the spider is being closed. See `spider_closed` signal for more info.

`scrapy.signals.stats_spider_closed(spider, reason, spider_stats)`

Sent right after the stats spider is closed. You can use this signal to collect resources, but not to add any more stats as the stats spider has already been closed (use `stats_spider_closing` for that instead).

**Parameters**

- **spider** (*str*) – the stats spider just closed
- **reason** (*dict*) – the reason why the spider was closed. See `spider_closed` signal for more info.
- **spider\_stats** – the stats of the spider just closed.

## 4.3 Sending e-mail

Although Python makes sending e-mails relatively easy via the `smtplib` library, Scrapy provides its own facility for sending e-mails which is very easy to use and it's implemented using `Twisted non-blocking IO`, to avoid interfering with the non-blocking IO of the crawler. It also provides a simple API for sending attachments and it's very easy to configure, with a few *settings* <topics-email-settings>.

### 4.3.1 Quick example

Here's a quick example of how to send an e-mail (without attachments):

```
from scrapy.mail import MailSender
```

```
mailer = MailSender()
```

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

### 4.3.2 MailSender class reference

`MailSender` is the preferred class to use for sending emails from Scrapy, as it uses `Twisted non-blocking IO`, like the rest of the framework.

`MailSender(smtphost=None, mailfrom=None, smtpuser=None, smtppass=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.
- **smtppass** (*str*) – the SMTP pass for authentication.
- **smtpport** (*int*) – the SMTP port to connect to

`scrapy.mail.send(to, subject, body, cc=None, attachs=())`  
Send email to the given recipients. Emits the `mail_sent` signal.

**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

### 4.3.3 Mail settings

These settings define the default constructor values of the `MailSender` class, and can be used to configure e-mail notifications in your project without writing any code (for those extensions and code that uses `MailSender`).

**MAIL\_FROM**

Default: `'scrapy@localhost'`

Sender email to use (`From:` header) for sending emails.

**MAIL\_HOST**

Default: `'localhost'`

SMTP host to use for sending emails.

**MAIL\_PORT**

Default: `25`

SMTP port to use for sending emails.

## MAIL\_USER

Default: None

User to use for SMTP authentication. If disabled no SMTP authentication will be performed.

## MAIL\_PASS

Default: None

Password to use for SMTP authentication, along with `MAIL_USER`.

### 4.3.4 Mail signals

`scrapy.mail.mail_sent` (*to*, *subject*, *body*, *cc*, *attachs*, *msg*)

Emitted by `MailSender.send()` after an email has been sent.

#### 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
- **msg** (`MIMEMultipart` or `MIMENonMultipart`) – the generated message

## 4.4 Telnet Console

Scrapy comes with a built-in telnet console for inspecting and controlling a Scrapy running process. The telnet console is just a regular python shell running inside the Scrapy process, so you can do literally anything from it.

The telnet console is a *built-in Scrapy extension* which comes enabled by default, but you can also disable it if you want. For more information about the extension itself see *Telnet console extension*.

### 4.4.1 How to access the telnet console

The telnet console listens in the TCP port defined in the `TELNETCONSOLE_PORT` setting, which defaults to 6023. To access the console you need to type:

```
telnet localhost 6023
>>>
```

You need the telnet program which comes installed by default in Windows, and most Linux distros.

### 4.4.2 Available variables in the telnet console

The telnet console is like a regular Python shell running inside the Scrapy process, so you can do anything from it including importing new modules, etc.

However, the telnet console comes with some default variables defined for convenience:

Shortcut	Description
crawler	the Scrapy Crawler object ( <code>scrapy.crawler</code> )
engine	the Scrapy Engine object ( <code>scrapy.core.engine</code> )
spider	the spider object (only if there is a single spider opened)
slot	the engine slot (only if there is a single spider opened)
extensions	the Extension Manager ( <code>scrapy.project.crawler.extensions</code> )
stats	the Stats Collector ( <code>scrapy.stats.stats</code> )
settings	the Scrapy settings object ( <code>scrapy.conf.settings</code> )
est	print a report of the current engine status
prefs	for memory debugging (see <a href="#">Debugging memory leaks</a> )
p	a shortcut to the <code>pprint.pprint</code> function
hpy	for memory debugging (see <a href="#">Debugging memory leaks</a> )

### 4.4.3 Telnet console usage examples

Here are some example tasks you can do with the telnet console:

#### View engine status

You can use the `est()` method of the Scrapy engine to quickly show its state using the telnet console:

```
telnet localhost 6023
>>> est()
Execution engine status

time()-engine.start_time           : 21.3188259602
engine.is_idle()                   : False
engine.has_capacity()               : True
engine.scheduler.is_idle()         : False
len(engine.scheduler.pending_requests) : 1
engine.downloader.is_idle()        : False
len(engine.downloader.slots)       : 1
engine.scrapers.is_idle()          : False
len(engine.scrapers.slots)         : 1

Spider: <GayotSpider 'gayotcom' at 0x2dc2b10>
  engine.spider_is_idle(spider)      : False
  engine.slots[spider].closing       : False
  len(engine.scheduler.pending_requests[spider]) : 11504
  len(engine.downloader.slots[spider].queue) : 9
  len(engine.downloader.slots[spider].active) : 17
  len(engine.downloader.slots[spider].transferring) : 8
  engine.downloader.slots[spider].lastseen : 1311311093.61
  len(engine.scrapers.slots[spider].queue) : 0
  len(engine.scrapers.slots[spider].active) : 0
  engine.scrapers.slots[spider].active_size : 0
  engine.scrapers.slots[spider].itemproc_size : 0
  engine.scrapers.slots[spider].needs_backout() : False
```

## Pause, resume and stop the Scrapy engine

To pause:

```
telnet localhost 6023
>>> engine.pause()
>>>
```

To resume:

```
telnet localhost 6023
>>> engine.unpause()
>>>
```

To stop:

```
telnet localhost 6023
>>> engine.stop()
Connection closed by foreign host.
```

### 4.4.4 Telnet Console signals

`scrapy.telnet.update_telnet_vars` (*telnet\_vars*)

Sent just before the telnet console is opened. You can hook up to this signal to add, remove or update the variables that will be available in the telnet local namespace. In order to do that, you need to update the `telnet_vars` dict in your handler.

**Parameters** `telnet_vars` (*dict*) – the dict of telnet variables

### 4.4.5 Telnet settings

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

#### TELNETCONSOLE\_PORT

Default: [6023, 6073]

The port range to use for the etlnet console. If set to `None` or `0`, a dynamically assigned port is used.

#### TELNETCONSOLE\_HOST

Default: '0.0.0.0'

The interface the telnet console should listen on

## 4.5 Web Service

Scrapy comes with a built-in web service for monitoring and controlling a running crawler. The service exposes most resources using the [JSON-RPC 2.0](#) protocol, but there are also other (read-only) resources which just output JSON data.

Provides an extensible web service for managing a Scrapy process. It's enabled by the `WEBSERVICE_ENABLED` setting. The web server will listen in the port specified in `WEBSERVICE_PORT`, and will log to the file specified in `WEBSERVICE_LOGFILE`.

The web service is a *built-in Scrapy extension* which comes enabled by default, but you can also disable it if you're running tight on memory.

### 4.5.1 Web service resources

The web service contains several resources, defined in the `WEBSERVICE_RESOURCES` setting. Each resource provides a different functionality. See *Available JSON-RPC resources* for a list of resources available by default.

Although you can implement your own resources using any protocol, there are two kinds of resources bundled with Scrapy:

- Simple JSON resources - which are read-only and just output JSON data
- JSON-RPC resources - which provide direct access to certain Scrapy objects using the *JSON-RPC 2.0* protocol

#### Available JSON-RPC resources

These are the JSON-RPC resources available by default in Scrapy:

##### Crawler JSON-RPC resource

**class** `scrapy.contrib.webservice.crawler.CrawlerResource`  
Provides access to the main Crawler object that controls the Scrapy process.  
Available by default at: <http://localhost:6080/crawler>

##### Stats Collector JSON-RPC resource

**class** `scrapy.contrib.webservice.stats.StatsResource`  
Provides access to the Stats Collector used by the crawler.  
Available by default at: <http://localhost:6080/stats>

##### Spider Manager JSON-RPC resource

You can access the spider manager JSON-RPC resource through the *Crawler JSON-RPC resource* at: <http://localhost:6080/crawler/spiders>

##### Extension Manager JSON-RPC resource

You can access the extension manager JSON-RPC resource through the *Crawler JSON-RPC resource* at: <http://localhost:6080/crawler/spiders>

#### Available JSON resources

These are the JSON resources available by default:

### Engine status JSON resource

`class scrapy.contrib.webservice.enginestatus.EngineStatusResource`

Provides access to engine status metrics.

Available by default at: <http://localhost:6080/enginestatus>

## 4.5.2 Web service settings

These are the settings that control the web service behaviour:

### WEBSERVICE\_ENABLED

Default: `True`

A boolean which specifies if the web service will be enabled (provided its extension is also enabled).

### WEBSERVICE\_LOGFILE

Default: `None`

A file to use for logging HTTP requests made to the web service. If unset web the log is sent to standard scrapy log.

### WEBSERVICE\_PORT

Default: `[6080, 7030]`

The port range to use for the web service. If set to `None` or `0`, a dynamically assigned port is used.

### WEBSERVICE\_HOST

Default: `'0.0.0.0'`

The interface the web service should listen on

### WEBSERVICE\_RESOURCES

Default: `{}`

The list of web service resources enabled for your project. See *Web service resources*. These are added to the ones available by default in Scrapy, defined in the `WEBSERVICE_RESOURCES_BASE` setting.

### WEBSERVICE\_RESOURCES\_BASE

Default:

```
{
    'scrapy.contrib.webservice.crawler.CrawlerResource': 1,
    'scrapy.contrib.webservice.enginestatus.EngineStatusResource': 1,
    'scrapy.contrib.webservice.stats.StatsResource': 1,
}
```



The list of web service resources available by default in Scrapy. You shouldn't change this setting in your project, change `WEBSERVICE_RESOURCES` instead. If you want to disable some resource set its value to `None` in `WEBSERVICE_RESOURCES`.

### 4.5.3 Writing a web service resource

Web service resources are implemented using the Twisted Web API. See this [Twisted Web guide](#) for more information on Twisted web and Twisted web resources.

To write a web service resource you should subclass the `JsonResource` or `JsonRpcResource` classes and implement the `renderGET` method.

**class scrapy.webservice.JsonResource**

A subclass of `twisted.web.resource.Resource` that implements a JSON web service resource. See

**ws\_name**

The name by which the Scrapy web service will know this resource, and also the path where this resource will listen. For example, assuming Scrapy web service is listening on <http://localhost:6080/> and the `ws_name` is `'resource1'` the URL for that resource will be:

<http://localhost:6080/resource1/>

**class scrapy.webservice.JsonRpcResource** (*crawler, target=None*)

This is a subclass of `JsonResource` for implementing JSON-RPC resources. JSON-RPC resources wrap Python (Scrapy) objects around a JSON-RPC API. The resource wrapped must be returned by the `get_target()` method, which returns the target passed in the constructor by default

**get\_target()**

Return the object wrapped by this JSON-RPC resource. By default, it returns the object passed on the constructor.

### 4.5.4 Examples of web service resources

#### StatsResource (JSON-RPC resource)

```
from scrapy.webservice import JsonRpcResource
from scrapy.stats import stats

class StatsResource(JsonRpcResource):

    ws_name = 'stats'

    def __init__(self, crawler):
        JsonRpcResource.__init__(self, crawler, stats)
```

#### EngineStatusResource (JSON resource)

```
from scrapy.webservice import JsonResource
from scrapy.utils.engine import get_engine_status

class EngineStatusResource(JsonResource):

    ws_name = 'enginestatus'

    def __init__(self, crawler, spider_name=None):
```

```
JsonResource.__init__(self, crawler)
self._spider_name = spider_name
self.isLeaf = spider_name is not None

def render_GET(self, txrequest):
    status = get_engine_status(self.crawler.engine)
    if self._spider_name is None:
        return status
    for sp, st in status['spiders'].items():
        if sp.name == self._spider_name:
            return st

def getChild(self, name, txrequest):
    return EngineStatusResource(name, self.crawler)
```

## 4.5.5 Example of web service client

### scrapy-ws.py script

```
#!/usr/bin/env python
"""
Example script to control a Scrapy server using its JSON-RPC web service.

It only provides a reduced functionality as its main purpose is to illustrate
how to write a web service client. Feel free to improve or write you own.

Also, keep in mind that the JSON-RPC API is not stable. The recommended way for
controlling a Scrapy server is through the execution queue (see the "queue"
command).

"""

import sys, optparse, urllib
from urlparse import urljoin

from scrapy.utils.jsonrpc import jsonrpc_client_call, JsonRpcError
from scrapy.utils.py26 import json

def get_commands():
    return {
        'help': cmd_help,
        'stop': cmd_stop,
        'list-available': cmd_list_available,
        'list-running': cmd_list_running,
        'list-resources': cmd_list_resources,
        'get-global-stats': cmd_get_global_stats,
        'get-spider-stats': cmd_get_spider_stats,
    }

def cmd_help(args, opts):
    """help - list available commands"""
    print "Available commands:"
    for _, func in sorted(get_commands().items()):
        print "  ", func.__doc__

def cmd_stop(args, opts):
```

```

"""stop <spider> - stop a running spider"""
jsonrpc_call(opts, 'crawler/engine', 'close_spider', args[0])

def cmd_list_running(args, opts):
    """list-running - list running spiders"""
    for x in jsonrpc_call(opts, 'crawler/engine/open_spiders'):
        print x

def cmd_list_available(args, opts):
    """list-available - list name of available spiders"""
    for x in jsonrpc_call(opts, 'crawler/spiders', 'list'):
        print x

def cmd_list_resources(args, opts):
    """list-resources - list available web service resources"""
    for x in json_get(opts, '')['resources']:
        print x

def cmd_get_spider_stats(args, opts):
    """get-spider-stats <spider> - get stats of a running spider"""
    stats = jsonrpc_call(opts, 'stats', 'get_stats', args[0])
    for name, value in stats.items():
        print "%-40s %s" % (name, value)

def cmd_get_global_stats(args, opts):
    """get-global-stats - get global stats"""
    stats = jsonrpc_call(opts, 'stats', 'get_stats')
    for name, value in stats.items():
        print "%-40s %s" % (name, value)

def get_wsurl(opts, path):
    return urljoin("http://%s:%s/" % (opts.host, opts.port), path)

def jsonrpc_call(opts, path, method, *args, **kwargs):
    url = get_wsurl(opts, path)
    return jsonrpc_client_call(url, method, *args, **kwargs)

def json_get(opts, path):
    url = get_wsurl(opts, path)
    return json.loads(urllib.urlopen(url).read())

def parse_opts():
    usage = "%prog [options] <command> [arg] ..."
    description = "Scrapy web service control script. Use '%prog help' " \
        "to see the list of available commands."
    op = optparse.OptionParser(usage=usage, description=description)
    op.add_option("-H", dest="host", default="localhost", \
        help="Scrapy host to connect to")
    op.add_option("-P", dest="port", type="int", default=6080, \
        help="Scrapy port to connect to")
    opts, args = op.parse_args()
    if not args:
        op.print_help()
        sys.exit(2)
    cmdname, cmdargs, opts = args[0], args[1:], opts
    commands = get_commands()
    if cmdname not in commands:
        sys.stderr.write("Unknown command: %s\n\n" % cmdname)

```

```
        cmd_help(None, None)
        sys.exit(1)
    return commands[cmdname], cmdargs, opts

def main():
    cmd, args, opts = parse_opts()
    try:
        cmd(args, opts)
    except IndexError:
        print cmd.__doc__
    except JsonRpcError, e:
        print str(e)
        if e.data:
            print "Server Traceback below:"
            print e.data

if __name__ == '__main__':
    main()
```

**Logging** Understand the simple logging facility provided by Scrapy.

**Stats Collection** Collect statistics about your scraping crawler.

**Sending e-mail** Send email notifications when certain events occur.

**Telnet Console** Inspect a running crawler using a built-in Python console.

**Web Service** Monitor and control a crawler using a web service.

---

## Solving specific problems

---

### 5.1 Frequently Asked Questions

#### 5.1.1 How does Scrapy compare to BeautifulSoup or lxml?

[BeautifulSoup](#) and [lxml](#) are libraries for parsing HTML and XML. Scrapy is an application framework for writing web spiders that crawl web sites and extract data from them.

Scrapy provides a built-in mechanism for extracting data (called *selectors*) but you can easily use [BeautifulSoup](#) (or [lxml](#)) instead, if you feel more comfortable working with them. After all, they're just parsing libraries which can be imported and used from any Python code.

In other words, comparing [BeautifulSoup](#) (or [lxml](#)) to Scrapy is like comparing [jinja2](#) to [Django](#).

#### 5.1.2 What Python versions does Scrapy support?

Scrapy runs in Python 2.5, 2.6 and 2.7. But it's recommended you use Python 2.6 or above, since the Python 2.5 standard library has a few bugs in their URL handling libraries. Some of these Python 2.5 bugs not only affect Scrapy but any user code, such as spiders.

#### 5.1.3 Does Scrapy work with Python 3.0?

No, and there are no plans to port Scrapy to Python 3.0 yet. At the moment, Scrapy works with Python 2.5, 2.6 and 2.7.

**See also:**

*What Python versions does Scrapy support?*

#### 5.1.4 Did Scrapy “steal” X from Django?

Probably, but we don't like that word. We think [Django](#) is a great open source project and an example to follow, so we've used it as an inspiration for Scrapy.

We believe that, if something is already done well, there's no need to reinvent it. This concept, besides being one of the foundations for open source and free software, not only applies to software but also to documentation, procedures, policies, etc. So, instead of going through each problem ourselves, we choose to copy ideas from those projects that have already solved them properly, and focus on the real problems we need to solve.

We'd be proud if Scrapy serves as an inspiration for other projects. Feel free to steal from us!

### 5.1.5 Does Scrapy work with HTTP proxies?

Yes. Support for HTTP proxies is provided (since Scrapy 0.8) through the HTTP Proxy downloader middleware. See [HttpProxyMiddleware](#).

### 5.1.6 Scrapy crashes with: ImportError: No module named win32api

You need to install [pywin32](#) because of [this Twisted bug](#).

### 5.1.7 How can I simulate a user login in my spider?

See *Using FormRequest.from\_response() to simulate a user login*.

### 5.1.8 Does Scrapy crawl in breath-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.squeue.PickleFifoDiskQueue'
SCHEDULER_MEMORY_QUEUE = 'scrapy.squeue.FifoMemoryQueue'
```

### 5.1.9 My Scrapy crawler has memory leaks. What can I do?

See *Debugging memory leaks*.

Also, Python has a builtin memory leak issue which is described in *Leaks without leaks*.

### 5.1.10 How can I make Scrapy consume less memory?

See previous question.

### 5.1.11 Can I use Basic HTTP Authentication in my spiders?

Yes, see [HttpAuthMiddleware](#).

### 5.1.12 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.

### 5.1.13 Where can I find some example Scrapy projects?

See *Examples*.

### 5.1.14 Can I run a spider without creating a project?

Yes. You can use the `runspider` command. For example, if you have a spider written in a `my_spider.py` file you can run it with:

```
scrapy runspider my_spider.py
```

See `runspider` command for more info.

### 5.1.15 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 message are thrown by the Offsite Spider Middleware, which is a spider middleware (enabled by default) whose purpose is to filter out requests to domains outside the ones covered by the spider.

For more info see: `OffsiteMiddleware`.

### 5.1.16 What is the recommended way to deploy a Scrapy crawler in production?

See *Scrapy Service* (`scrapyd`).

### 5.1.17 Can I use JSON for large exports?

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

### 5.1.18 Can I return (Twisted) deferreds from signal handlers?

Some signals support returning deferreds from their handlers, others don’t. See the *Built-in signals reference* to know which ones.

### 5.1.19 What does the response status code 999 means?

999 is a custom reponse status code used by Yahoo sites to throttle requests. Try slowing down the crawling speed by using a download delay of 2 (or higher) in your spider:

```
class MySpider(CrawlSpider):  
    name = 'myspider'  
  
    DOWNLOAD_DELAY = 2  
  
    # [ ... rest of the spider code ... ]
```

Or by setting a global download delay in your project with the `DOWNLOAD_DELAY` setting.

### 5.1.20 Can I call `pdb.set_trace()` from my spiders to debug them?

Yes, but you can also use the Scrapy shell which allows you too quickly analyze (and even modify) the response being processed by your spider, which is, quite often, more useful than plain old `pdb.set_trace()`.

For more info see *Invoking the shell from spiders to inspect responses*.

### 5.1.21 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 -t json
```

To dump into a CSV file:

```
scrapy crawl myspider -o items.csv -t csv
```

To dump into a XML file:

```
scrapy crawl myspider -o items.xml -t xml
```

For more information see *Feed exports*

### 5.1.22 What's this huge cryptic `__VIEWSTATE` parameter used in some forms?

The `__VIEWSTATE` parameter is used in sites built with ASP.NET/VB.NET. For more info on how it works see [this page](#). Also, here's an [example spider](#) which scrapes one of these sites.

### 5.1.23 What's the best way to parse big XML/CSV data feeds?

Parsing big feeds with XPath selectors can be problematic since they need to build the DOM of the entire feed in memory, and this can be quite slow and consume a lot of memory.

In order to avoid parsing all the entire feed at once in memory, you can use the functions `xmliter` and `csviter` from `scrapy.utils.iterators` module. In fact, this is what the feed spiders (see *Spiders*) use under the cover.

### 5.1.24 Does Scrapy manage cookies automatically?

Yes, Scrapy receives and keeps track of cookies sent by servers, and sends them back on subsequent requests, like any regular web browser does.

For more info see *Requests and Responses* and *CookiesMiddleware*.

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

Enable the `COOKIES_DEBUG` setting.

### 5.1.26 How can I instruct a spider to stop itself?

Raise the `CloseSpider` exception from a callback. For more info see: `CloseSpider`.



### 5.1.27 How can I prevent my Scrapy bot from 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.

Here are some tips to keep in mind when dealing with these kind 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.
- is 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](#).

If you are still unable to prevent your bot getting banned, consider contacting [commercial support](#).

## 5.2 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.

### 5.2.1 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

### 5.2.2 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.

## 5.3 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.

---

### 5.3.1 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](#).

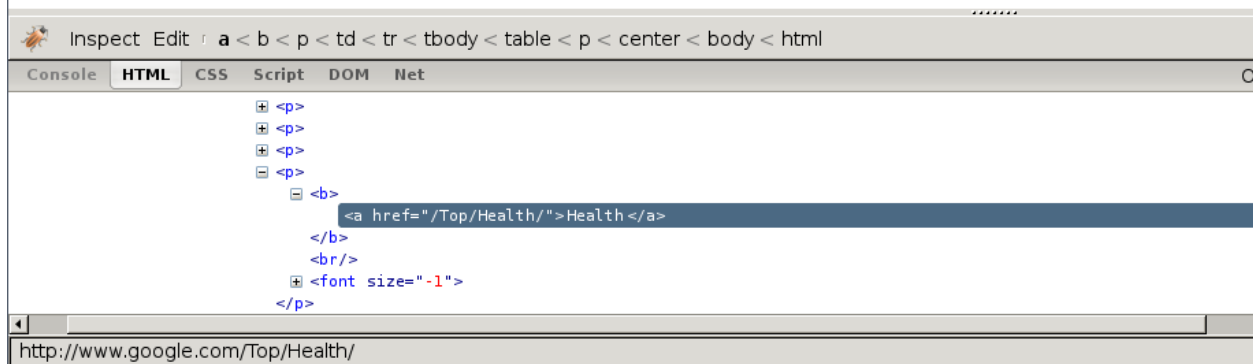
In this example, we'll show how to use [Firebug](#) to scrape data from the [Google Directory](#), which contains the same data as the [Open Directory Project](#) used in the [tutorial](#) but with a different face.

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.

The web organized by topic into categories.

<b>Arts</b> <a href="#">Movies</a> , <a href="#">Music</a> , <a href="#">Television</a> , ...	<b>Home</b> <a href="#">Consumers</a> , <a href="#">Homeowners</a> , <a href="#">Family</a> , ...	<b>Region</b> <a href="#">Asia</a> , <a href="#">Europe</a>
<b>Business</b> <a href="#">Industries</a> , <a href="#">Finance</a> , <a href="#">Jobs</a> , ...	<b>Kids and Teens</b> <a href="#">Computers</a> , <a href="#">Entertainment</a> , <a href="#">School</a> , ...	<b>Science</b> <a href="#">Biology</a> , <a href="#">Physics</a> , ...
<b>Computers</b> <a href="#">Hardware</a> , <a href="#">Internet</a> , <a href="#">Software</a> , ...	<b>News</b> <a href="#">Media</a> , <a href="#">Newspapers</a> , <a href="#">Current Events</a> , ...	<b>Shopping</b> <a href="#">Autos</a> , <a href="#">Clothing</a> , ...
<b>Games</b> <a href="#">Board</a> , <a href="#">Roleplaying</a> , <a href="#">Video</a> , ...	<b>Recreation</b> <a href="#">Food</a> , <a href="#">Outdoors</a> , <a href="#">Travel</a> , ...	<b>Society</b> <a href="#">Issues</a> , <a href="#">People</a> , ...
<b>Health</b> <a href="#">Alternative</a> , <a href="#">Fitness</a> , <a href="#">Medicine</a> , ...	<b>Reference</b> <a href="#">Education</a> , <a href="#">Libraries</a> , <a href="#">Maps</a> , ...	<b>Sports</b> <a href="#">Basketball</a> , <a href="#">Baseball</a> , ...



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:

[Environmental Health](#) (359)      [Products and Shopping](#) (61)      [Weight Loss](#) (357)  
[Fitness](#) (961)      [Professions](#) (1692)      [Women's Health](#) (764)  
[Healthcare Industry](#) (6380)

Related Categories:

- [Business > Business Services > Consulting > Medical and Life Sciences](#) (321)
- [Kids and Teens > Health](#) (1160)
- [Recreation > Humor > Medical](#) (26)
- [Science > Social Sciences > Communication > Health Communication](#) (3)
- [Shopping > Health](#) (7391)
- [Society > Issues > Health](#) (2592)

Web Pages	Viewing in Google PageRank order	Vi
<a href="http://www.webmd.com/">WebMD</a> - <a href="http://www.webmd.com/">http://www.webmd.com/</a> A health resources for consumers, physicians, nurses, and educators. Includes news, chat forums, health quizzes and consumer product updates.		
<a href="http://www.hon.ch/">Health On the Net Foundation</a> - <a href="http://www.hon.ch/">http://www.hon.ch/</a> Guides lay persons and non-medical users and medical practitioners to useful and reliable online medical and health information.		
<a href="http://www.bbc.co.uk/health/">BBC Health</a> - <a href="http://www.bbc.co.uk/health/">http://www.bbc.co.uk/health/</a> Features current news plus archives, guides by subject, "Ask a Doctor" inquiry feature, a searchable conditions database, medical advice, and more.		
<a href="http://www.aolhealth.com/">AOL Health</a> - <a href="http://www.aolhealth.com/">http://www.aolhealth.com/</a>		

Inspect Edit **a** < font < td < tr < tbody < table < form < body < html

Console HTML CSS Script DOM Net

```

<table width="100%" cellspacing="0" cellpadding="1" border="0">
  <tbody>
    <tr valign="top">
      <td width="6%">
        <td>
          <font face="arial,sans-serif">
            <a href="http://www.webmd.com/">WebMD</a>
            <font size="-1" color="#6f6f6f">
              <br/>
              <font size="-1"> A health resources for consumers, physicians, nurses, and educators. Includes news, chat forums, health quizzes and consumer product updates.</font>
            </font>
          </td>
        </td>
      </tr>
    </tbody>
  </table>
  
```

Done

As expected, the subcategories contain links to other subcategories, and also links to actual websites, which is the purpose of the directory.

### 5.3.2 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\.google\.com/[A-Z][a-zA-Z_/>+]$
```

So, based on that regular expression we can create the first crawling rule:

```
Rule(SgmlLinkExtractor(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:

```
from scrapy.contrib.linkextractors.sgml import SgmlLinkExtractor
from scrapy.contrib.spiders import CrawlSpider, Rule
```

```

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

    rules = (
        Rule(SgmlLinkExtractor(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

```

### 5.3.3 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 *XPath selectors*. We'll also use the *Scrapy shell* to test those XPath's and make sure they work as we expect.

Shopping > Health (7391)  
Society > Issues > Health (2592)

Web Pages	Viewing in Google PageRank order	View in alph
<a href="http://www.webmd.com/">WebMD</a> - http://www.webmd.com/ A health resources for consumers, physicians, nurses, and educators. Includes news, chat forums, health quizzes and consumer product		
<a href="http://www.hon.ch/">Health On the Net Foundation</a> - http://www.hon.ch/ Guides lay persons and non-medical users and medical practitioners to useful and reliable online medical and health information. Provide		
<a href="http://www.bbc.co.uk/health/">BBC Health</a> - http://www.bbc.co.uk/health/ Features current news plus archives, guides by subject, "Ask a Doctor" inquiry feature, a searchable conditions database, message board		
<a href="http://www.aolhealth.com/">AOL Health</a> - http://www.aolhealth.com/ Find advice, information about diseases and drugs, fitness tips, and news items.		

Inspect Edit td <tr> tbody <table> <form> <body> <html>

Console HTML CSS Script DOM Net Options

```

<table width="100%" cellspacing="0" cellpadding="0" border="0">
  <table width="100%" cellspacing="0" cellpadding="1" border="0">
    <tbody>
      <tr valign="top">
        <td width="6%">
          <nobr>
            <a href="http://www.google.com/intl/en/dirhelp.html#pagerank">
          </nobr>
        </td>
      </tr>
    </tbody>
  </table>
</table>

```

Done

```

[<HtmlXPathSelector (a) xpath=//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a>,
<HtmlXPathSelector (a) xpath=//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a>,
<HtmlXPathSelector (a) xpath=//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a>,
<HtmlXPathSelector (a) xpath=//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a>,
<HtmlXPathSelector (a) xpath=//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a>,
<HtmlXPathSelector (a) xpath=//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a>]

In [5]: hxs.x('//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td//a').extract()
Out[5]:
[u'<a href="http://www.webmd.com/">WebMD</a>',
u'<a href="http://www.hon.ch/">Health On the Net Foundation</a>',
u'<a href="http://www.bbc.co.uk/health/">BBC Health</a>',
u'<a href="http://www.aolhealth.com/">AOL Health</a>',
u'<a href="http://www.intelihealth.com/">InteliHealth</a>',
u'<a href="http://www.judgehealth.org.uk/">Judge: Web Sites for Health</a>']

In [6]:

```

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 XPath's.

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`).

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:

```
def parse_category(self, response):
    hxs = HtmlXPathSelector(response)

    # The path to website links in directory page
    links = hxs.select('//td[descendant::a[contains(@href, "#pagerank")]]/following-sibling::td/font')

    for link in links:
        item = DirectoryItem()
        item['name'] = link.select('a/text()').extract()
        item['url'] = link.select('a/@href').extract()
        item['description'] = link.select('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.

or tags which Therefer in page HTML sources may on Firebug inspects the live DOM

## 5.4 Debugging memory leaks

In Scrapy, objects such as Requests, Responses and Items have a finite lifetime: they are created, used for a while, and finally destroyed.

From all those objects, the Request is probably the one with the longest lifetime, as it stays waiting in the Scheduler queue until it's time to process it. For more info see *Architecture overview*.

As these Scrapy objects have a (rather long) lifetime, there is always the risk of accumulating them in memory without releasing them properly and thus causing what is known as a “memory leak”.

To help debugging memory leaks, Scrapy provides a built-in mechanism for tracking objects references called *trackref*, and you can also use a third-party library called *Guppy* for more advanced memory debugging (see below for more info). Both mechanisms must be used from the *Telnet Console*.

### 5.4.1 Common causes of memory leaks

It happens quite often (sometimes by accident, sometimes on purpose) that the Scrapy developer passes objects referenced in Requests (for example, using the `meta` attribute or the request callback function) and that effectively bounds the lifetime of those referenced objects to the lifetime of the Request. This is, by far, the most common cause of memory leaks in Scrapy projects, and a quite difficult one to debug for newcomers.

In big projects, the spiders are typically written by different people and some of those spiders could be “leaking” and thus affecting the rest of the other (well-written) spiders when they get to run concurrently, which, in turn, affects the whole crawling process.

At the same time, it’s hard to avoid the reasons that cause these leaks without restricting the power of the framework, so we have decided not to restrict the functionality but provide useful tools for debugging these leaks, which quite often consist in an answer to the question: *which spider is leaking?*.

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, if you’re allocating resources on `spider_opened` but not releasing them on `spider_closed`.

### 5.4.2 Debugging memory leaks with `trackref`

`trackref` is a module provided by Scrapy to debug the most common cases of memory leaks. It basically tracks the references to all live Requests, Responses, Item and Selector objects.

To activate the `trackref` module, enable the `TRACK_REFS` setting. It only imposes a minor performance impact, so it should be OK to use it, even in production environments.

Once you have `trackref` enabled, 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
XPathSelector          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 do have leaks, 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 like this (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.XPathSelector`
- `scrapy.spider.BaseSpider`

- `scrapy.selector.document.Libxml2Document`

## A real example

Let's see a concrete example of an 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 which one is the nasty spider (without knowing it a-priori, of course) by using the `trackref` tool.

After the crawler is running for a few minutes and we notice its memory usage has grown a lot, we can enter its telnet console and check the live references:

```
>>> prefs()
Live References

SomenastySpider          1   oldest: 15s ago
HtmlResponse             3890  oldest: 265s ago
XPathSelector            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. So let's check the oldest response:

```
>>> from scrapy.utils.trackref import get_oldest
>>> r = get_oldest('HtmlResponse')
>>> r.url
'http://www.somenastyspider.com/product.php?pid=123'
```

There it is. By looking at the URL of the oldest response we can see it belongs to the `somenastyspider.com` spider. We can now go and check the code of that spider to discover the nasty line that is generating the leaks (passing response references inside requests).

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

```
>>> from scrapy.utils.trackref import iter_all
>>> [r.url for r in iter_all('HtmlResponse')]
['http://www.somenastyspider.com/product.php?pid=123',
 'http://www.somenastyspider.com/product.php?pid=584',
 ...]
```

## Too many spiders?

If your project has too many spiders, 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, using:

```
>>> from scrapy.spider import BaseSpider
>>> prefs(ignore=BaseSpider)
```

Won't show any live references to spiders.



## scrapy.utils.trackref module

Here are the functions available in the `trackref` module.

**class** `scrapy.utils.trackref.object_ref`

Inherit from this class (instead of `object`) if you want to track live instances with the `trackref` module.

`scrapy.utils.trackref.print_live_refs(class_name, ignore=NoneType)`

Print a report of live references, grouped by class name.

**Parameters** `ignore` (*class or classes tuple*) – if given, all objects from the specified class (or tuple of classes) will be ignored.

`scrapy.utils.trackref.get_oldest(class_name)`

Return the oldest object alive with the given class name, or `None` if none is found. Use `print_live_refs()` first to get a list of all tracked live objects per class name.

`scrapy.utils.trackref.iter_all(class_name)`

Return an iterator over all objects alive with the given class name, or `None` if none is found. Use `print_live_refs()` first to get a list of all tracked live objects per class name.

## 5.4.3 Debugging memory leaks with Guppy

`trackref` provides a very convenient mechanism for tracking down memory leaks, but it only keeps track of the objects that are more likely to cause memory leaks (Requests, Responses, Items, and Selectors). However, there are other cases where the memory leaks could come from other (more or less obscure) objects. If this is your case, and you can't find your leaks using `trackref`, you still have another resource: the [Guppy library](#).

If you use `setuptools`, you can install Guppy with the following command:

```
easy_install guppy
```

The telnet console also comes with a built-in shortcut (`hpy`) for accessing Guppy heap objects. Here's an example to view all Python objects available in the heap using Guppy:

```
>>> x = hpy.heap()
>>> x.bytype
Partition of a set of 297033 objects. Total size = 52587824 bytes.
  Index  Count    %      Size    % Cumulative  % Type
    0    22307    8  16423880   31  16423880   31 dict
    1   122285   41  12441544   24  28865424   55 str
    2    68346   23   5966696   11  34832120   66 tuple
    3     227    0   5836528   11  40668648   77 unicode
    4    2461    1   2222272    4  42890920   82 type
    5   16870    6   2024400    4  44915320   85 function
    6   13949    5   1673880    3  46589200   89 types.CodeType
    7   13422    5   1653104    3  48242304   92 list
    8    3735    1   1173680    2  49415984   94 _sre.SRE_Pattern
    9    1209    0    456936    1  49872920   95 scrapy.http.headers.Headers
<1676 more rows. Type e.g. '_.more' to view.>
```

You can see that most space is used by dicts. Then, if you want to see from which attribute those dicts are referenced, you could do:

```
>>> x.bytype[0].byvia
Partition of a set of 22307 objects. Total size = 16423880 bytes.
  Index  Count    %      Size    % Cumulative  % Referred Via:
    0   10982   49   9416336   57   9416336   57 '.__dict__'
    1    1820    8   2681504   16  12097840   74 '.__dict__', '.func_globals'
```

```
2    3097   14  1122904    7  13220744   80
3     990    4   277200    2  13497944   82 "['cookies']"
4     987    4   276360    2  13774304   84 "['cache']"
5     985    4   275800    2  14050104   86 "['meta']"
6     897    4   251160    2  14301264   87 '[2]'
7        1    0   196888    1  14498152   88 "['moduleDict']", "['modules']"
8     672    3   188160    1  14686312   89 "['cb_kwargs']"
9      27    0   155016    1  14841328   90 '[1]'
<333 more rows. Type e.g. '_.more' to view.>
```

As you can see, the Guppy module is very powerful but also requires some deep knowledge about Python internals. For more info about Guppy, refer to the [Guppy documentation](#).

### 5.4.4 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.*

This problem will be fixed in future Scrapy releases, where we plan to adopt a new process model and run spiders in a pool of recyclable sub-processes.

## 5.5 Downloading Item Images

Scrapy provides an *item pipeline* for downloading images attached to a particular item, for example, when you scrape products and also want to download their images locally.

This pipeline, called the Images Pipeline and implemented in the [ImagesPipeline](#) class, provides a convenient way for downloading and storing images locally with some additional features:

- Convert all downloaded images to a common format (JPG) and mode (RGB)
- Avoid re-downloading images which were downloaded recently
- Thumbnail generation
- Check images width/height to make sure they meet a minimum constraint

This pipeline also keeps an internal queue of those images which are currently being scheduled for download, and connects those items that arrive containing the same image, to that queue. This avoids downloading the same image more than once when it's shared by several items.

The [Python Imaging Library](#) is used for thumbnailing and normalizing images to JPEG/RGB format, so you need to install that library in order to use the images pipeline.

### 5.5.1 Using the Images Pipeline

The typical workflow, when using the `ImagesPipeline` goes like this:

1. In a Spider, you scrape an item and put the URLs of its images into a `image_urls` field.
2. The item is returned from the spider and goes to the item pipeline.
3. When the item reaches the `ImagesPipeline`, the URLs in the `image_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 images have finish downloading (or fail for some reason).
4. When the images are downloaded another field (`images`) will be populated with the results. This field will contain a list of dicts with information about the images downloaded, such as the downloaded path, the original scraped url (taken from the `image_urls` field) , and the image checksum. The images in the list of the `images` field will retain the same order of the original `image_urls` field. If some image failed downloading, an error will be logged and the image won’t be present in the `images` field.

### 5.5.2 Usage example

In order to use the image pipeline you just need to *enable it* and define an item with the `image_urls` and `images` fields:

```
from scrapy.item import Item

class MyItem(Item):

    # ... other item fields ...
    image_urls = Field()
    images = Field()
```

If you need something more complex and want to override the custom images pipeline behaviour, see [Implementing your custom Images Pipeline](#).

### 5.5.3 Enabling your Images Pipeline

To enable your images pipeline you must first add it to your project `ITEM_PIPELINES` setting:

```
ITEM_PIPELINES = ['scrapy.contrib.pipeline.images.ImagesPipeline']
```

And set the `IMAGES_STORE` setting to a valid directory 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 example:

```
IMAGES_STORE = '/path/to/valid/dir'
```

### 5.5.4 Images Storage

File system is currently the only officially supported storage, but there is also (undocumented) support for [Amazon S3](#).

## File system storage

The images are stored in files (one per image), 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
- `full` is a sub-directory to separate full images from thumbnails (if used). For more info see *Thumbnail generation*.

## 5.5.5 Additional features

### Image expiration

The Image Pipeline avoids downloading images that were downloaded recently. To adjust this retention delay use the `IMAGES_EXPIRES` setting, which specifies the delay in number of days:

```
# 90 days of delay for image expiration
IMAGES_EXPIRES = 90
```

### Thumbnail generation

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.

For example:

```
IMAGES_THUMBS = {
    'small': (50, 50),
    'big': (270, 270),
}
```

When you use this feature, the Images Pipeline will create thumbnails of the each specified size with this format:

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

Where:

- `<size_name>` is the one specified in the `IMAGES_THUMBS` dictionary keys (`small`, `big`, etc)
- `<image_id>` is the [SHA1 hash](#) of the image url

Example of image files stored using `small` and `big` thumbnail names:

```
<IMAGES_STORE>/full/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/small/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
<IMAGES_STORE>/thumbs/big/63bbfea82b8880ed33cdb762aa11fab722a90a24.jpg
```

The first one is the full image, as downloaded from the site.

### Filtering out small images

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:

```
IMAGES_MIN_HEIGHT = 110
IMAGES_MIN_WIDTH = 110
```

Note: these size constraints don't affect thumbnail generation at all.

By default, there are no size constraints, so all images are processed.

## 5.5.6 Implementing your custom Images Pipeline

Here are the methods that you should override in your custom Images Pipeline:

```
class scrapy.contrib.pipeline.images.ImagesPipeline
```

```
    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 must override the `get_media_requests()` method and return a Request for each image URL:

```
def get_media_requests(self, item, info):
    for image_url in item['image_urls']:
        yield Request(image_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, image\_info\_or\_failure) where:

- success is a boolean which is True if the image was downloaded successfully or False if it failed for some reason
- image\_info\_or\_error is a dict containing the following keys (if success is True) or a `TwistedFailure` if there was a problem.
  - url - the url where the image was downloaded from. This is the url of the request returned from the `get_media_requests()` method.
  - path - the path (relative to `IMAGES_STORE`) where the image 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/7d97e98f8af710c7e7fe703abc8f639e0ee507c4.jpg',
  'url': 'http://www.example.com/images/product1.jpg'}),
 (True,
 {'checksum': 'b9628c4ab9b595f72f280b90c4fd093d',
  'path': 'full/1ca5879492b8fd606df1964ea3c1e2f4520f076f.jpg',
  'url': 'http://www.example.com/images/product2.jpg'}),
 (False,
 Failure(...))]
```

By default the `get_media_requests()` method returns `None` which means there are no images to download for the item.

**item\_completed** (*results, items, info*)

The `ImagesPipeline.item_completed()` method called when all image 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 image paths (passed in results) in the `image_paths` item field, and we drop the item if it doesn't contain any images:

```
from scrapy.exceptions import DropItem

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
```

By default, the `item_completed()` method returns the item.

## 5.5.7 Custom Images pipeline example

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

```
from scrapy.contrib.pipeline.images import ImagesPipeline
from scrapy.exceptions import DropItem
from scrapy.http import Request

class MyImagesPipeline(ImagesPipeline):

    def get_media_requests(self, item, info):
        for image_url in item['image_urls']:
            yield Request(image_url)

    def item_completed(self, results, item, info):
        image_paths = [x['path'] for ok, x in results if ok]
        if not image_paths:
            raise DropItem("Item contains no images")
        item['image_paths'] = image_paths
        return item
```

## 5.6 Ubuntu packages

New in version 0.10.

*Insophia* publishes apt-gettable packages which are generally fresher than those in Ubuntu, and more stable too since they're continuously built from [Github repo](#) (master & stable branches) and so they contain the latest bug fixes.

To use the packages, just add the following line to your `/etc/apt/sources.list`, and then run `aptitude update` and `aptitude install scrapy-0.13`:

```
deb http://archive.scrapy.org/ubuntu DISTRO main
```

Replacing `DISTRO` with the name of your Ubuntu release, which you can get with command:

```
lsb_release -cs
```

Supported Ubuntu releases are: karmic, lucid, maverick, natty, oneiric, precise.

For Ubuntu Precise (12.04):

```
deb http://archive.scrapy.org/ubuntu precise main
```

For Ubuntu Oneiric (11.10):

```
deb http://archive.scrapy.org/ubuntu oneiric main
```

For Ubuntu Natty (11.04):

```
deb http://archive.scrapy.org/ubuntu natty main
```

For Ubuntu Maverick (10.10):

```
deb http://archive.scrapy.org/ubuntu maverick main
```

For Ubuntu Lucid (10.04):

```
deb http://archive.scrapy.org/ubuntu lucid main
```

For Ubuntu Karmic (9.10):

```
deb http://archive.scrapy.org/ubuntu karmic main
```

**Warning:** Please note that these packages are updated frequently, and so if you find you can't download the packages, try updating your apt package lists first, e.g., with `apt-get update` or `aptitude update`.

The public GPG key used to sign these packages can be imported into you APT keyring as follows:

```
curl -s http://archive.scrapy.org/ubuntu/archive.key | sudo apt-key add -
```

## 5.7 Scrapy Service (scrapyd)

New in version 0.10.

Scrapy comes with a built-in service, called “Scrapyd”, which allows you to deploy (aka. upload) your projects and control their spiders using a JSON web service.

### 5.7.1 Projects and versions

Scrapyd can manage multiple projects and each project can have multiple versions uploaded, but only the latest one will be used for launching new spiders.

A common (and useful) convention to use for the version name is the revision number of the version control tool you're using to track your Scrapy project code. For example: `r23`. The versions are not compared alphabetically but using a smarter algorithm (the same `distutils` uses) so `r10` compares greater to `r9`, for example.

### 5.7.2 How Scrapyd works

Scrapyd is an application (typically run as a daemon) that continually polls for spiders that need to run.

When a spider needs to run, a process is started to crawl the spider:

```
scrapy crawl myspider
```

Scrapyd also runs multiple processes in parallel, allocating them in a fixed number of slots given by the `max_proc` and `max_proc_per_cpu` options, starting as many processes as possible to handle the load.

In addition to dispatching and managing processes, Scrapyd provides a *JSON web service* to upload new project versions (as eggs) and schedule spiders. This feature is optional and can be disabled if you want to implement your own custom Scrapyd. The components are pluggable and can be changed, if you're familiar with the *Twisted Application Framework* which Scrapyd is implemented in.

Starting from 0.11, Scrapyd also provides a minimal *web interface*.

### 5.7.3 Starting Scrapyd

Scrapyd is implemented using the standard *Twisted Application Framework*. To start the service, use the `extras/scrapyd.tac` file provided in the Scrapy distribution, like this:

```
twistd -ny extras/scrapyd.tac
```

That should get your Scrapyd started.

Or, if you want to start Scrapyd from inside a Scrapy project you can use the `server` command, like this:

```
scrapy server
```

### 5.7.4 Installing Scrapyd

How to deploy Scrapyd on your servers depends on the platform you're using. Scrapy comes with Ubuntu packages for Scrapyd ready for deploying it as a system service, to ease the installation and administration, but you can create packages for other distribution or operating systems (including Windows). If you do so, and want to contribute them, send a message to [scrapy-developers@googlegroups.com](mailto:scrapy-developers@googlegroups.com) and say hi. The community will appreciate it.

#### Installing Scrapyd in Ubuntu

When deploying Scrapyd, it's very useful to have a version already packaged for your system. For this reason, Scrapyd comes with Ubuntu packages ready to use in your Ubuntu servers.

So, if you plan to deploy Scrapyd on a Ubuntu server, just add the Ubuntu repositories as described in *Ubuntu packages* and then run:



```
aptitude install scrapyd-X.YY
```

Where `X.YY` is the Scrapy version, for example: `0.14`.

This will install Scrapyd in your Ubuntu server creating a `scrapy` user which Scrapyd will run as. It will also create some directories and files that are listed below:

#### **`/etc/scrapyd`**

Scrapyd configuration files. See *Scrapy Configuration file*.

#### **`/var/log/scrapyd/scrapyd.log`**

Scrapyd main log file.

#### **`/var/log/scrapyd/scrapyd.out`**

The standard output captured from Scrapyd process and any sub-process spawned from it.

#### **`/var/log/scrapyd/scrapyd.err`**

The standard error captured from Scrapyd and any sub-process spawned from it. Remember to check this file if you're having problems, as the errors may not get logged to the `scrapyd.log` file.

#### **`/var/log/scrapyd/project`**

Besides the main service log file, Scrapyd stores one log file per crawling process in:

```
/var/log/scrapyd/PROJECT/SPIDER/ID.log
```

Where `ID` is a unique id for the run.

#### **`/var/lib/scrapyd/`**

Directory used to store data files (uploaded eggs and spider queues).

### **5.7.5 Scrapyd Configuration file**

Scrapyd searches for configuration files in the following locations, and parses them in order with the latest ones taking more priority:

- `/etc/scrapyd/scrapyd.conf` (Unix)
- `c:\scrapyd\scrapyd.conf` (Windows)
- `/etc/scrapyd/conf.d/*` (in alphabetical order, Unix)
- `scrapyd.conf`

The configuration file supports the following options (see default values in the *example*).

### **http\_port**

The TCP port where the HTTP JSON API will listen. Defaults to 6800.

### **max\_proc**

The maximum number of concurrent Scrapy process that will be started. If unset or 0 it will use the number of cpus available in the system multiplied by the value in `max_proc_per_cpu` option. Defaults to 0.

### **max\_proc\_per\_cpu**

The maximum number of concurrent Scrapy process that will be started per cpu. Defaults to 4.

### **debug**

Whether debug mode is enabled. Defaults to `off`. When debug mode is enabled the full Python traceback will be returned (as plain text responses) when there is an error processing a JSON API call.

### **eggs\_dir**

The directory where the project eggs will be stored.

### **dbs\_dir**

The directory where the project databases will be stored (this includes the spider queues).

### **logs\_dir**

The directory where the Scrapy processes logs will be stored.

### **logs\_to\_keep**

The number of logs to keep per spider. Defaults to 5.

### **runner**

The module that will be used for launching sub-processes. You can customize the Scrapy processes launched from Scrapyd by using your own module.

### **application**

A function that returns the (Twisted) Application object to use. This can be used if you want to extend Scrapyd by adding and removing your own components and services.

For more info see [Twisted Application Framework](#)

## Example configuration file

Here is an example configuration file with all the defaults:

```
[scrapyd]
eggs_dir      = eggs
logs_dir      = logs
logs_to_keep  = 5
dbs_dir       = dbs
max_proc      = 0
max_proc_per_cpu = 4
http_port     = 6800
debug         = off
runner        = scrapyd.runner
application   = scrapyd.app.application
```

### 5.7.6 Deploying your project

Deploying your project into a Scrapy server typically involves two steps:

1. building a [Python egg](#) of your project. This is called “eggifying” your project. You’ll need to install [setuptools](#) for this. See [Egg caveats](#) below.
2. uploading the egg to the Scrapy server

The simplest way to deploy your project is by using the [deploy](#) command, which automates the process of building the egg uploading it using the Scrapy HTTP JSON API.

The [deploy](#) command supports multiple targets (Scrapy servers that can host your project) and each target supports multiple projects.

Each time you deploy a new version of a project, you can name it for later reference.

#### Show and define targets

To see all available targets type:

```
scrapy deploy -l
```

This will return a list of available targets and their URLs. For example:

```
scrapyd                http://localhost:6800/
```

You can define targets by adding them to your project’s `scrapy.cfg` file, or any other supported location like `~/.scrapy.cfg`, `/etc/scrapy.cfg`, or `c:\scrapy\scrapy.cfg` (in Windows).

Here’s an example of defining a new target `scrapyd2` with restricted access through HTTP basic authentication:

```
[deploy:scrapyd2]
url = http://scrapyd.mydomain.com/api/scrapyd/
username = john
password = secret
```

---

**Note:** The [deploy](#) command also supports `netrc` for getting the credentials.

---

Now, if you type `scrapy deploy -l` you’ll see:

```
scrapyd          http://localhost:6800/
scrapyd2         http://scrapyd.mydomain.com/api/scrapyd/
```

### See available projects

To see all available projects in a specific target use:

```
scrapy deploy -L scrapyd
```

It would return something like this:

```
project1
project2
```

### Deploying a project

Finally, to deploy your project use:

```
scrapy deploy scrapyd -p project1
```

This will eggify your project and upload it to the target, printing the JSON response returned from the Scrapy server. If you have a `setup.py` file in your project, that one will be used. Otherwise a `setup.py` file will be created automatically (based on a simple template) that you can edit later.

After running that command you will see something like this, meaning your project was uploaded successfully:

```
Deploying myproject-1287453519 to http://localhost:6800/addversion.json
Server response (200):
{"status": "ok", "spiders": ["spider1", "spider2"]}
```

By default `scrapy deploy` uses the current timestamp for generating the project version, as you can see in the output above. However, you can pass a custom version with the `--version` option:

```
scrapy deploy scrapyd -p project1 --version 54
```

Also, if you use Mercurial for tracking your project source code, you can use `HG` for the version which will be replaced by the current Mercurial revision, for example `r382`:

```
scrapy deploy scrapyd -p project1 --version HG
```

And, if you use Git for tracking your project source code, you can use `GIT` for the version which will be replaced by the SHA1 of current Git revision, for example `b0582849179d1de7bd86eaa7201ea3cda4b5651f`:

```
scrapy deploy scrapyd -p project1 --version GIT
```

Support for other version discovery sources may be added in the future.

Finally, if you don't want to specify the target, project and version every time you run `scrapy deploy` you can define the defaults in the `scrapy.cfg` file. For example:

```
[deploy]
url = http://scrapyd.mydomain.com/api/scrapyd/
username = john
password = secret
project = project1
version = HG
```

This way, you can deploy your project just by using:

```
scrapy deploy
```

## Local settings

Sometimes, while your working on your projects, you may want to override your certain settings with certain local settings that shouldn't be deployed to Scrapyd, but only used locally to develop and debug your spiders.

One way to deal with this is to have a `local_settings.py` at the root of your project (where the `scrapy.cfg` file resides) and add these lines to the end of your project settings:

```
try:
    from local_settings import *
except ImportError:
    pass
```

`scrapy deploy` won't deploy anything outside the project module so the `local_settings.py` file won't be deployed.

Here's the directory structure, to illustrate:

```
scrapy.cfg
local_settings.py
myproject/
  __init__.py
  settings.py
  spiders/
  ...
```

## Egg caveats

There are some things to keep in mind when building eggs of your Scrapy project:

- make sure no local development settings are included in the egg when you build it. The `find_packages` function may be picking up your custom settings. In most cases you want to upload the egg with the default project settings.
- you shouldn't use `__file__` in your project code as it doesn't play well with eggs. Consider using `pkgutil.get_data()` instead.
- be careful when writing to disk in your project (in any spider, extension or middleware) as Scrapyd will probably run with a different user which may not have write access to certain directories. If you can, avoid writing to disk and always use `tempfile` for temporary files.

## 5.7.7 Scheduling a spider run

To schedule a spider run:

```
$ curl http://localhost:6800/schedule.json -d project=myproject -d spider=spider2
{"status": "ok", "jobid": "26d1b1a6d6f111e0be5c001e648c57f8"}
```

For more resources see: [JSON API reference](#) for more available resources.

## 5.7.8 Web Interface

New in version 0.11.

Scrapyd comes with a minimal web interface (for monitoring running processes and accessing logs) which can be accessed at <http://localhost:6800/>

## 5.7.9 JSON API reference

The following section describes the available resources in Scrapyd JSON API.

### addversion.json

Add a version to a project, creating the project if it doesn't exist.

- Supported Request Methods: POST
- Parameters:
  - project (string, required) - the project name
  - version (string, required) - the project version
  - egg (file, required) - a Python egg containing the project's code

Example request:

```
$ curl http://localhost:6800/addversion.json -F project=myproject -F version=r23 -F egg=@myproject.egg
```

Example response:

```
{"status": "ok", "spiders": 3}
```

### schedule.json

Schedule a spider run.

- Supported Request Methods: POST
- Parameters:
  - project (string, required) - the project name
  - spider (string, required) - the spider name
  - setting (string, optional) - a scrapy setting to use when running the spider
  - any other parameter is passed as spider argument

Example request:

```
$ curl http://localhost:6800/schedule.json -d project=myproject -d spider=somespider
```

Example response:

```
{"status": "ok"}
```

Example request passing a spider argument (arg1) and a setting (DOWNLOAD\_DELAY):

```
$ curl http://localhost:6800/schedule.json -d project=myproject -d spider=somespider -d setting=DOWNLOAD_DELAY=1
```

### listprojects.json

Get the list of projects uploaded to this Scrapy server.

- Supported Request Methods: GET
- Parameters: none

Example request:

```
$ curl http://localhost:6800/listprojects.json
```

Example response:

```
{"status": "ok", "projects": ["myproject", "otherproject"]}
```

### listversions.json

Get the list of versions available for some project. The versions are returned in order, the last one is the currently used version.

- Supported Request Methods: GET
- Parameters:
  - project (string, required) - the project name

Example request:

```
$ curl http://localhost:6800/listversions.json?project=myproject
```

Example response:

```
{"status": "ok", "versions": ["r99", "r156"]}
```

### listspiders.json

Get the list of spiders available in the last version of some project.

- Supported Request Methods: GET
- Parameters:
  - project (string, required) - the project name

Example request:

```
$ curl http://localhost:6800/listspiders.json?project=myproject
```

Example response:

```
{"status": "ok", "spiders": ["spider1", "spider2", "spider3"]}
```

### delversion.json

Delete a project version. If there are no more versions available for a given project, that project will be deleted too.

- Supported Request Methods: POST
- Parameters:

- `project` (string, required) - the project name
- `version` (string, required) - the project version

Example request:

```
$ curl http://localhost:6800/delversion.json -d project=myproject -d version=r99
```

Example response:

```
{"status": "ok"}
```

### **delproject.json**

Delete a project and all its uploaded versions.

- Supported Request Methods: POST
- Parameters:
  - `project` (string, required) - the project name

Example request:

```
$ curl http://localhost:6800/delproject.json -d project=myproject
```

Example response:

```
{"status": "ok"}
```

## **5.8 Jobs: pausing and resuming crawls**

Sometimes, for big sites, it's desirable to pause crawls and be able to resume them later.

Scrapy supports this functionality out of the box by providing the following facilities:

- a scheduler that persists scheduled requests on disk
- a duplicates filter that persists visited requests on disk
- an extension that keeps some spider state (key/value pairs) persistent between batches

### **5.8.1 Job directory**

To enable persistence support you just need to define a *job directory* through the `JOBDIR` setting. This directory will be for storing all required data to keep the state of a single job (ie. a spider run). It's important to note that this directory must not be shared by different spiders, or even different jobs/runs of the same spider, as it's meant to be used for storing the state of a *single* job.

### **5.8.2 How to use it**

To start a spider with persistence supported enabled, run it like this:

```
scrapy crawl somespider -s JOBDIR=crawls/somespider-1
```

Then, you can stop the spider safely at any time (by pressing Ctrl-C or sending a signal), and resume it later by issuing the same command:



```
scrapy crawl somespider -s JOBDIR=crawls/somespider-1
```

### 5.8.3 Keeping persistent state between batches

Sometimes you'll want to keep some persistent spider state between pause/resume batches. You can use the `spider.state` attribute for that, which should be a dict. There's a built-in extension that takes care of serializing, storing and loading that attribute from the job directory, when the spider starts and stops.

Here's an example of a callback that uses the spider state (other spider code is omitted for brevity):

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

### 5.8.4 Persistence gotchas

There are a few things to keep in mind if you want to be able to use the Scrapy persistence support:

#### Cookies expiration

Cookies may expire. So, if you don't resume your spider quickly the requests scheduled may no longer work. This won't be an issue if you spider doesn't rely on cookies.

#### Request serialization

Requests must be serializable by the *pickle* module, in order for persistence to work, so you should make sure that your requests are serializable.

The most common issue here is to use `lambda` functions on request callbacks that can't be persisted.

So, for example, this won't work:

```
def some_callback(self, response):
    somearg = 'test'
    return Request('http://www.example.com', callback=lambda r: self.other_callback(r, somearg))

def other_callback(self, response, somearg):
    print "the argument passed is:", somearg
```

But this will:

```
def some_callback(self, response):
    somearg = 'test'
    return Request('http://www.example.com', meta={'somearg': somearg})

def other_callback(self, response):
    somearg = response.meta['somearg']
    print "the argument passed is:", somearg
```

**Frequently Asked Questions** Get answers to most frequently asked questions.

**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 Item Images* Download static images associated with your scraped items.

*Ubuntu packages* Install latest Scrapy packages easily on Ubuntu

*Scrapy Service (scrapyd)* Deploying your Scrapy project in production.

*Jobs: pausing and resuming crawls* Learn how to pause and resume crawls for large spiders.

---

## Extending Scrapy

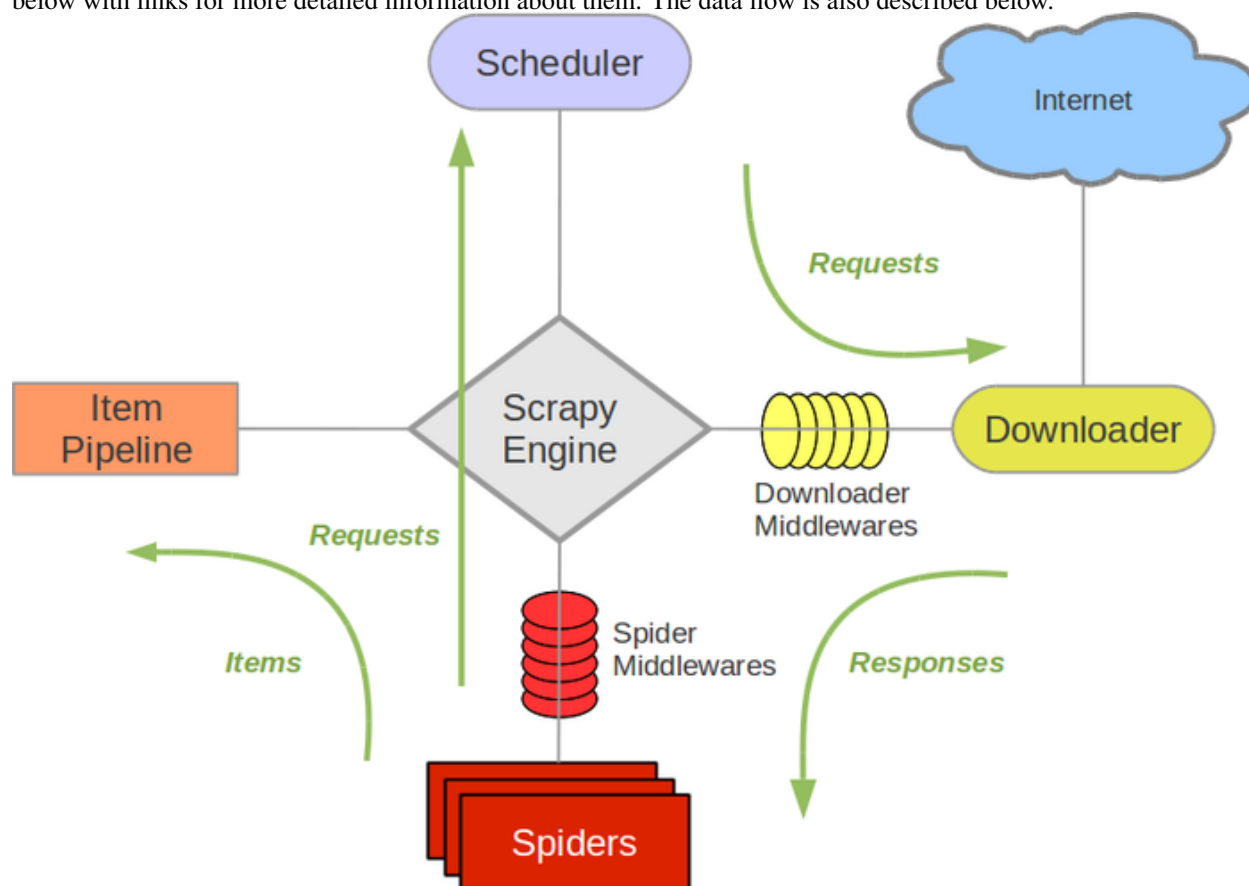
---

### 6.1 Architecture overview

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

#### 6.1.1 Overview

The following diagram shows an overview of the Scrapy architecture with its components and an outline of the data flow that takes place inside the system (shown by the 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.



## 6.1.2 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 below 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 URLs (requests) to follow. Each spider is able to handle a specific domain (or group of domains). 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. They provide a convenient mechanism for extending Scrapy functionality by plugging custom code. 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). They provide a convenient mechanism for extending Scrapy functionality by plugging custom code. For more information see *Spider Middleware*.

## 6.1.3 Data flow

The data flow in Scrapy is controlled by the Engine, and goes like this:

1. The Engine opens a domain, locates the Spider that handles that domain, and asks the spider for the first URLs to crawl.
2. The Engine gets the first URLs to crawl from the Spider and schedules them in the Scheduler, as Requests.

3. The Engine asks the Scheduler for the next URLs to crawl.
4. The Scheduler returns the next URLs to crawl to the Engine and the Engine sends them to the Downloader, passing through the Downloader Middleware (request direction).
5. Once the page finishes downloading the Downloader generates a Response (with that page) and sends it to the Engine, passing through the Downloader Middleware (response direction).
6. The Engine receives the Response from the Downloader and sends it to the Spider for processing, passing through the Spider Middleware (input direction).
7. The Spider processes the Response and returns scraped Items and new Requests (to follow) to the Engine.
8. The Engine sends scraped Items (returned by the Spider) to the Item Pipeline and Requests (returned by spider) to the Scheduler
9. The process repeats (from step 2) until there are no more requests from the Scheduler, and the Engine closes the domain.

## 6.1.4 Event-driven networking

Scrapy is written with [Twisted](#), a popular event-driven networking framework for Python. Thus, it's implemented using a non-blocking (aka asynchronous) code for concurrency.

For more information about asynchronous programming and Twisted see these links:

- [Asynchronous Programming with Twisted](#)
- [Twisted - hello, asynchronous programming](#)

## 6.2 Downloader Middleware

The downloader middleware is a framework of hooks into Scrapy's request/response processing. It's a light, low-level system for globally altering Scrapy's requests and responses.

### 6.2.1 Activating a downloader middleware

To activate a downloader middleware component, add it to the `DOWNLOADER_MIDDLEWARES` setting, which is a dict whose keys are the middleware class paths and their values are the middleware orders.

Here's an example:

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

The `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.

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 off-site middleware:

```
DOWNLOADER_MIDDLEWARES = {
    'myproject.middlewares.CustomDownloaderMiddleware': 543,
    'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware': None,
}
```

Finally, keep in mind that some middlewares may need to be enabled through a particular setting. See each middleware documentation for more info.

## 6.2.2 Writing your own downloader middleware

Writing your own downloader middleware is easy. Each middleware component is a single Python class that defines one or more of the following methods:

```
class scrapy.contrib.downloadermiddleware.DownloaderMiddleware
```

**process\_request** (*request*, *spider*)

This method is called for each request that goes through the download middleware.

`process_request()` should return either `None`, a `Response` object, or a `Request` object.

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 request or exception middleware, or the appropriate download function; it'll return that `Response`. Response middleware is always called on every `Response`.

If it returns a `Request` object, the returned request will be rescheduled (in the Scheduler) to be downloaded in the future. The callback of the original request will always be called. If the new request has a callback it will be called with the response downloaded, and the output of that callback will then be passed to the original callback. If the new request doesn't have a callback, the response downloaded will be just passed to the original request callback.

If it returns an `IgnoreRequest` exception, the entire request will be dropped completely and its callback never called.

### Parameters

- **request** (`Request` object) – the request being processed
- **spider** (`BaseSpider` object) – the spider for which this request is intended

**process\_response** (*request*, *response*, *spider*)

`process_response()` should return a `Response` 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 pipeline.

If it returns an `IgnoreRequest` exception, the response will be dropped completely and its callback never called.

### Parameters

- **request** (is a `Request` object) – the request that originated the response
- **response** – the response being processed
- **spider** (`BaseSpider` 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.

`process_exception()` should return either `None`, `Response` or `Request` object.

If it returns `None`, Scrapy will continue processing this exception, executing any other exception middleware, until no middleware is left and the default exception handling kicks in.

If it returns a `Response` object, the response middleware kicks in, and won't bother calling any other exception middleware.

If it returns a `Request` object, the returned request is used to instruct an immediate redirection. The original request won't finish until the redirected request is completed. This stops the `process_exception()` middleware the same as returning `Response` would do.

#### Parameters

- **request** (is a `Request` object) – the request that generated the exception
- **exception** (an `Exception` object) – the raised exception
- **spider** (`BaseSpider` object) – the spider for which this request is intended

### 6.2.3 Built-in downloader middleware reference

This page describes all downloader middleware components that come with Scrapy. For information on how to use them and how to write your own downloader middleware, see the [downloader middleware usage guide](#).

For a list of the components enabled by default (and their orders) see the `DOWNLOADER_MIDDLEWARES_BASE` setting.

#### CookiesMiddleware

**class** `scrapy.contrib.downloadermiddleware.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`

#### 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 [diningcity] INFO: Spider opened
2011-04-06 14:35:10-0300 [diningcity] DEBUG: Sending cookies to: <GET http://www.diningcity.com/neth
    Cookie: clientlanguage_nl=en_EN
2011-04-06 14:35:14-0300 [diningcity] DEBUG: Received cookies from: <200 http://www.diningcity.com/n
    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=/
2011-04-06 14:49:50-0300 [diningcity] DEBUG: Crawled (200) <GET http://www.diningcity.com/netherlands
[...]
```

### DefaultHeadersMiddleware

**class** scrapy.contrib.downloadermiddleware.defaultheaders.**DefaultHeadersMiddleware**

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

### DownloadTimeoutMiddleware

**class** scrapy.contrib.downloadermiddleware.downloadtimeout.**DownloadTimeoutMiddleware**

This middleware sets the download timeout for requests specified in the `DOWNLOAD_TIMEOUT` setting.

### HttpAuthMiddleware

**class** scrapy.contrib.downloadermiddleware.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:

```
class SomeIntranetSiteSpider(CrawlSpider):

    http_user = 'someuser'
    http_pass = 'somepass'
    name = 'intranet.example.com'

    # .. rest of the spider code omitted ...
```

### HttpCacheMiddleware

**class** scrapy.contrib.downloadermiddleware.httpcache.**HttpCacheMiddleware**

This middleware provides low-level cache to all HTTP requests and responses. 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 HTTP cache 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.

Scrapy ships with two storage backends for the HTTP cache middleware:

- *File system backend (default)*
- *DBM storage backend*



You can change the storage backend with the `HTTPCACHE_STORAGE` setting. Or you can also implement your own backend.

### File system backend (default)

By default, the `HttpCacheMiddleware` uses a file system storage with the following structure:

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. To use it (instead of the default filesystem backend) set `HTTPCACHE_STORAGE` to `scrapy.contrib.httpcache.DbmCacheStorage`.

By default, it uses the `anydbm` module, but you can change it with the `HTTPCACHE_DBM_MODULE` setting.

### 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.contrib.downloadermiddleware.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.

### HttpCompressionMiddleware

**class** scrapy.contrib.downloadermiddleware.httpcompression.**HttpCompressionMiddleware**  
This middleware allows compressed (gzip, deflate) traffic to be sent/received from web sites.

### ChunkedTransferMiddleware

**class** scrapy.contrib.downloadermiddleware.chunked.**ChunkedTransferMiddleware**  
This middleware adds support for *chunked transfer encoding*

### HttpProxyMiddleware

New in version 0.8.

**class** scrapy.contrib.downloadermiddleware.httpproxy.**HttpProxyMiddleware**  
This middleware sets the HTTP proxy to use for requests, by setting the `proxy` meta value to *Request* objects.

Like the Python standard library modules *urllib* and *urllib2*, it obeys the following environment variables:

- `http_proxy`
- `https_proxy`
- `no_proxy`

## RedirectMiddleware

**class** scrapy.contrib.downloadermiddleware.redirect.**RedirectMiddleware**

This middleware handles redirection of requests based on response status and meta-refresh html tag.

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`
- `REDIRECT_MAX_METAREFRESH_DELAY`

If `Request.meta` contains the `dont_redirect` key, the request will be ignored by this middleware.

### 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.

**REDIRECT\_MAX\_METAREFRESH\_DELAY** Default: `100`

The maximum meta-refresh delay (in seconds) to follow the redirection.

## RetryMiddleware

**class** scrapy.contrib.downloadermiddleware.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`

About HTTP errors to consider:

You may want to remove 400 from `RETRY_HTTP_CODES`, if you stick to the HTTP protocol. It's included by default because it's a common code used to indicate server overload, which would be something we want to retry. If `Request.meta` contains the `dont_retry` key, 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, 503, 504, 400, 408]

Which HTTP response codes to retry. Other errors (DNS lookup issues, connections lost, etc) are always retried.

### RobotsTxtMiddleware

**class** scrapy.contrib.downloadermiddleware.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.

**Warning:** Keep in mind that, if you crawl using multiple concurrent requests per domain, Scrapy could still download some forbidden pages if they were requested before the robots.txt file was downloaded. This is a known limitation of the current robots.txt middleware and will be fixed in the future.

### DownloaderStats

**class** scrapy.contrib.downloadermiddleware.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.contrib.downloadermiddleware.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.

## 6.3 Spider Middleware

The spider middleware is a framework of hooks into Scrapy's spider processing mechanism where you can plug custom functionality to process the requests that are sent to *Spiders* for processing and to process the responses and items that are generated from spiders.

### 6.3.1 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:

```
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.

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 builtin 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:

```
SPIDER_MIDDLEWARES = {
    'myproject.middlewares.CustomSpiderMiddleware': 543,
    'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware': None,
}
```

Finally, keep in mind that some middlewares may need to be enabled through a particular setting. See each middleware documentation for more info.

### 6.3.2 Writing your own spider middleware

Writing your own spider middleware is easy. Each middleware component is a single Python class that defines one or more of the following methods:

```
class scrapy.contrib.spidermiddleware.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 handled 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** – the response being processed
- **spider** (`BaseSpider` 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` or `Item` objects.

#### Parameters

- **response** (class:~*scrapy.http.Response* object) – the response which generated this output from the spider
- **result** (an iterable of `Request` or `Item` objects) – the result returned by the spider
- **spider** (`BaseSpider` 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` 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** (`scrapy.spider.BaseSpider` object) – the spider which raised the exception

### 6.3.3 Built-in spider middleware reference

This page describes all spider middleware components that come with Scrapy. For information on how to use them and how to write your own spider middleware, see the *spider middleware usage guide*.

For a list of the components enabled by default (and their orders) see the `SPIDER_MIDDLEWARES_BASE` setting.

#### DepthMiddleware

**class** `scrapy.contrib.spidermiddleware.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.contrib.spidermiddleware.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.

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

```
class MySpider(CrawlSpider):
    handle_httpstatus_list = [404]
```

The `handle_httpstatus_list` key of `Request.meta` can also be used to specify which response codes to allow on a per-request basis.

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](#).

## OffsiteMiddleware

**class** scrapy.contrib.spidermiddleware.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.

When your spider returns a request for a domain not belonging to those covered by the spider, this middleware will log a debug message similar to this one:

```
DEBUG: Filtered offsite request to 'www.othersite.com': <GET http://www.othersite.com/some/page.
```

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.contrib.spidermiddleware.referer.**RefererMiddleware**

Populates Request referer field, based on the Response which originated it.

## UrlLengthMiddleware

**class** scrapy.contrib.spidermiddleware.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.

## 6.4 Extensions

The extensions framework provides a mechanism for inserting your own custom functionality into Scrapy.

Extensions are just regular classes that are instantiated at Scrapy startup, when extensions are initialized.

### 6.4.1 Extension settings

Extensions use the *Scrapy settings* to manage their settings, just like any other Scrapy code.

It is customary for extensions to prefix their settings with their own name, to avoid collision with existing (and future) extensions. For example, an hypothetical extension to handle [Google Sitemaps](#) would use settings like `GOOGLE-SITEMAP_ENABLED`, `GOOGLESITEMAP_DEPTH`, and so on.

### 6.4.2 Loading & activating extensions

Extensions are loaded and activated at startup by instantiating a single instance of the extension class. Therefore, all the extension initialization code must be performed in the class constructor (`__init__` method).

To make an extension available, add it to the `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.contrib.corestats.CoreStats': 500,
    'scrapy.webservice.WebService': 500,
    'scrapy.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. Extensions orders are not as important as middleware orders though, and they are typically irrelevant, ie. it doesn't matter in which order the extensions are loaded because they don't depend on each other [1].

However, this feature can be exploited if you need to add an extension which depends on other extensions already loaded.

[1] This is why the `EXTENSIONS_BASE` setting in Scrapy (which contains all built-in extensions enabled by default) defines all the extensions with the same order (500).

### 6.4.3 Available, enabled and disabled extensions

Not all available extensions will be enabled. Some of them usually depend on a particular setting. For example, the HTTP Cache extension is available by default but disabled unless the `HTTPCACHE_ENABLED` setting is set.

### 6.4.4 Accessing enabled extensions

Even though it's not usually needed, you can access extension objects through the *Extension Manager* which is populated when extensions are loaded. For example, to access the `WebService` extension:



```
from scrapy.project import extensions
webservice_extension = extensions.enabled['WebService']
```

### 6.4.5 Writing your own extension

Writing your own extension is easy. Each extension is a single Python class which doesn't need to implement any particular method.

All extension initialization code must be performed in the class constructor (`__init__` method). If that method raises the `NotConfigured` exception, the extension will be disabled. Otherwise, the extension will be enabled.

Let's take a look at the following example extension which just logs a message every time a domain/spider is opened and closed:

```
from scrapy.xlib.pydispatch import dispatcher
from scrapy import signals

class SpiderOpenCloseLogging(object):

    def __init__(self):
        dispatcher.connect(self.spider_opened, signal=signals.spider_opened)
        dispatcher.connect(self.spider_closed, signal=signals.spider_closed)

    def spider_opened(self, spider):
        log.msg("opened spider %s" % spider.name)

    def spider_closed(self, spider):
        log.msg("closed spider %s" % spider.name)
```

### 6.4.6 Extension Manager

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.extension.ExtensionManager**

The Extension Manager is a singleton object, which is instantiated at module loading time and can be accessed like this:

```
from scrapy.project import extensions
```

**loaded**

A boolean which is True if extensions are already loaded or False if they're not.

**enabled**

A dict with the enabled extensions. The keys are the extension class names, and the values are the extension objects. Example:

```
>>> from scrapy.project import extensions
>>> extensions.load()
>>> print extensions.enabled
{'CoreStats': <scrapy.contrib.corestats.CoreStats object at 0x9e272ac>,
 'WebService': <scrapy.management.telnet.TelnetConsole instance at 0xa05670c>,
 ...
```

**disabled**

A dict with the disabled extensions. The keys are the extension class names, and the values are the extension class paths (because objects are never instantiated for disabled extensions). Example:

```
>>> from scrapy.project import extensions
>>> extensions.load()
>>> print extensions.disabled
{'MemoryDebugger': 'scrapy.contrib.memdebug.MemoryDebugger',
 'MyExtension': 'myproject.extensions.MyExtension',
 ...}
```

**load()**

Load the available extensions configured in the `EXTENSIONS` setting. On a standard run, this method is usually called by the Execution Manager, but you may need to call it explicitly if you're dealing with code outside Scrapy.

**reload()**

Reload the available extensions. See `load()`.

## 6.4.7 Built-in extensions reference

### General purpose extensions

#### Log Stats extension

```
class scrapy.contrib.logstats.LogStats
```

Log basic stats like crawled pages and scraped items.

#### Core Stats extension

```
class scrapy.contrib.corestats.CoreStats
```

Enable the collection of core statistics, provided the stats collection is enabled (see *Stats Collection*).

#### Web service extension

```
class scrapy.webservice.WebService
```

See *topics-webservice*.

#### Telnet console extension

```
class scrapy.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.contrib.memusage.MemoryUsage
```

---

**Note:** This extension does not work in Windows.

---

Allows monitoring the memory used by a Scrapy process and:

1. send a notification e-mail when it exceeds a certain value 2. terminate the Scrapy process 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 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`

## Memory debugger extension

```
class scrapy.contrib.memdebug.MemoryDebugger
```

An extension for debugging memory usage. It collects information about:

- objects uncollected by the Python garbage collector
- libxml2 memory leaks
- 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.contrib.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 `closespider_timeout`. If zero (or non set), spiders won't be closed by timeout.

### **CLOSESPIDER\_ITEMCOUNT** Default: 0

An integer which specifies a number of items. If the spider scrapes more than that amount of 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.contrib.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.contrib.debug.StackTraceDump
```

Dumps the stack trace and Scrapy engine status of a running process when a `SIGQUIT` or `SIGUSR2` signal is received. After the stack trace and engine status is dumped, the Scrapy process continues running normally.

The dump is sent to standard output.

This extension only works on POSIX-compliant platforms (ie. not 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.contrib.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).

*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* Add any custom functionality using [signals](#) and the Scrapy API



## 7.1 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*.

### 7.1.1 Request objects

```
class scrapy.http.Request (url[, method='GET', body, headers, cookies, meta, encoding='utf-8', priority=0, dont_filter=False, callback, 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
- **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).
- **cookies** (*dict*) – the request cookies. Example:

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

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 in the `Request.meta`.

Example of request without merging cookies:

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

For more info see [CookiesMiddleware](#).

- **encoding** (*string*) – the encoding of this request (defaults to `'utf-8'`). This encoding will be used to percent-encode the URL and to convert the body to `str` (if given as `unicode`).
- **priority** (*int*) – the priority of this request (defaults to 0). The priority is used by the scheduler to define the order used to process requests.
- **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`.
- **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.
- **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.

#### **url**

A string containing the URL of this request. Keep in mind that this attribute contains the escaped URL, so it can differ from the URL passed in the constructor.

This attribute is read-only. To change the URL of a `Request` use `replace()`.

#### **method**

A string representing the HTTP method in the request. This is guaranteed to be uppercase. Example: `"GET"`, `"POST"`, `"PUT"`, etc

#### **headers**

A dictionary-like object which contains the request headers.

#### **body**

A `str` that contains the request body.

This attribute is read-only. To change the body of a `Request` use `replace()`.

#### **meta**

A dict that contains arbitrary metadata for this request. This dict is empty for new `Requests`, and is usually populated by different Scrapy components (extensions, middlewares, etc). So the data contained in this dict depends on the extensions you have enabled.

See [Request.meta special keys](#) for a list of special meta keys recognized by Scrapy.

This dict is [shallow copied](#) when the request is cloned using the `copy()` or `replace()` methods, and can also be accessed, in your spider, from the `response.meta` attribute.

#### **copy()**

Return a new `Request` which is a copy of this `Request`. See also: [Passing additional data to callback functions](#).



**replace** ([*url, method, headers, body, cookies, meta, encoding, dont\_filter, callback, errback*])

Return a Request object with the same members, except for those members given new values by whichever keyword arguments are specified. The attribute `Request.meta` is copied by default (unless a new value is given in the `meta` argument). See also *Passing additional data to callback functions*.

## Passing additional data to callback functions

The callback of a request is a function that will be called when the response of that request is downloaded. The callback function will be called with the downloaded `Response` object as its first argument.

Example:

```
def parse_page1(self, response):
    return 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.log("Visited %s" % response.url)
```

In some cases you may be interested in passing arguments to those callback functions so you can receive the arguments later, in the second callback. You can use the `Request.meta` attribute for that.

Here's an example of how to pass an item using this mechanism, to populate different fields from different pages:

```
def parse_page1(self, response):
    item = MyItem()
    item['main_url'] = response.url
    request = 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
```

## 7.1.2 Request.meta special keys

The `Request.meta` attribute can contain any arbitrary data, but there are some special keys recognized by Scrapy and its built-in extensions.

Those are:

- `dont_redirect`
- `dont_retry`
- `handle_httpstatus_list`
- `dont_merge_cookies` (see `cookies` parameter of `Request` constructor)
- `redirect_urls`

## 7.1.3 Request subclasses

Here is the list of built-in `Request` subclasses. You can also subclass it to implement your own custom functionality.

## FormRequest objects

The `FormRequest` class extends the base `Request` with functionality for dealing with HTML forms. It uses the `ClientForm` library (bundled with Scrapy) to pre-populate form fields with form data from `Response` objects.

`class scrapy.http.FormRequest(url[, formdata, ...])`

The `FormRequest` class adds a new argument to the constructor. The remaining arguments are the same as for the `Request` class and are not documented here.

**Parameters** `formdata` (*dict or iterable of tuples*) – is a dictionary (or iterable of (key, value) tuples) containing HTML Form data which will be url-encoded and assigned to the body of the request.

The `FormRequest` objects support the following class method in addition to the standard `Request` methods:

`classmethod from_response(response[, formname=None, formnumber=0, formdata=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](#).

Keep in mind that this method is implemented using `ClientForm` whose 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()` (and `ClientForm`) 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. Otherwise, `formnumber` will be used for selecting the form.
- **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*) – Arguments to be passed directly to the `ClientForm click_request_data()` method. See `ClientForm` homepage for more info.
- **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.

## 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:

```
class LoginSpider(BaseSpider):
    name = 'example.com'
    start_urls = ['http://www.example.com/users/login.php']

    def parse(self, response):
        return [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.log("Login failed", level=log.ERROR)
            return

        # continue scraping with authenticated session...
```

## 7.1.4 Response objects

```
class scrapy.http.Response(url[, status=200, headers, body, flags])
```

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
- **headers** (*dict*) – the headers of this response. The dict values can be strings (for single valued headers) or lists (for multi-valued headers).
- **status** (*integer*) – the HTTP status of the response. Defaults to 200.
- **body** (*str*) – the response body. It must be str, not unicode, unless you're using a encoding-aware *Response subclass*, such as `TextResponse`.
- **meta** (*dict*) – the initial values for the `Response.meta` attribute. If given, the dict will be shallow copied.
- **flags** (*list*) – is a list containing the initial values for the `Response.flags` attribute. If given, the list will be shallow copied.

#### 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.

**body**

A str containing the body of this Response. Keep in mind that `Response.body` is always a str. If you want the unicode version use `TextResponse.body_as_unicode()` (only available in `TextResponse` and subclasses).

This attribute is read-only. To change the body of a Response use `replace()`.

**request**

The `Request` object that generated this response. This attribute is assigned in the Scrapy engine, after the response and the request have passed through all *Downloader Middlewares*. In particular, this means that:

- HTTP redirections will cause the original request (to the URL before redirection) to be assigned to the redirected response (with the final URL after redirection).
- `Response.request.url` doesn't always equal `Response.url`
- This attribute is only available in the spider code, and in the *Spider Middlewares*, but not in *Downloader Middlewares* (although you have the `Request` available there by other means) and handlers of the `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, meta, 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 (unless a new value is given in the `meta` argument).

## 7.1.5 Response subclasses

Here is the list of available built-in Response subclasses. You can also subclass the Response class to implement your own functionality.

## TextResponse objects

**class** scrapy.http.**TextResponse** (*url*[, *encoding*[, ...]])

**TextResponse** objects adds encoding capabilities to the base **Response** class, which is meant to be used only for binary data, such as images, sounds or any media file.

**TextResponse** objects support a new constructor argument, in addition to the base **Response** objects. The remaining functionality is the same as for the **Response** class and is not documented here.

**Parameters** **encoding** (*string*) – is a string which contains the encoding to use for this response. If you create a **TextResponse** object with a unicode body, it will be encoded using this encoding (remember the body attribute is always a string). If **encoding** is **None** (default value), the encoding will be looked up in the response headers and body instead.

**TextResponse** objects support the following attributes in addition to the standard **Response** ones:

### **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.

**TextResponse** objects support the following methods in addition to the standard **Response** ones:

### **body\_as\_unicode** ()

Returns the body of the response as unicode. This is equivalent to:

```
response.body.decode(response.encoding)
```

But **not** equivalent to:

```
unicode(response.body)
```

Since, in the latter case, you would be using you system default encoding (typically *ascii*) to convert the body to uniode, instead of the response encoding.

## HtmlResponse objects

**class** scrapy.http.**HtmlResponse** (*url*[, ...])

The **HtmlResponse** class is a subclass of **TextResponse** which adds encoding auto-discovering support by looking into the HTML *meta http-equiv* attribute. See **TextResponse.encoding**.

## XmlResponse objects

**class** scrapy.http.**XmlResponse** (*url*[, ...])

The **XmlResponse** class is a subclass of **TextResponse** which adds encoding auto-discovering support by looking into the XML declaration line. See **TextResponse.encoding**.

## 7.2 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*.

### 7.2.1 Designating the settings

When you use Scrapy, you have to tell it which settings you're using. You can do this by using an environment variable, `SCRAPY_SETTINGS_MODULE`.

The value of `SCRAPY_SETTINGS_MODULE` should be in Python path syntax, e.g. `myproject.settings`. Note that the settings module should be on the Python [import search path](#).

### 7.2.2 Populating the settings

Settings can be populated using different mechanisms, each of which having a different precedence. Here is the list of them in decreasing order of precedence:

1. Global overrides (most precedence)
2. Project settings module
3. Default settings per-command
4. Default global settings (less precedence)

These mechanisms are described in more detail below.

#### 1. Global overrides

Global overrides are the ones that take most precedence, and are usually populated by command-line options.

**Example::**

```
>>> from scrapy.conf import settings
>>> settings.overrides['LOG_ENABLED'] = True
```

You can also override one (or more) settings from command line using the `-s` (or `--set`) command line option.

**Example:**

```
scrapy crawl domain.com -s LOG_FILE=scrapy.log
```

#### 2. 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 example:: `myproject.settings`.

### 3. 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.

### 4. Default global settings

The global defaults are located in the `scrapy.settings.default_settings` module and documented in the *Built-in settings reference* section.

## 7.2.3 How to access settings

Here's an example of the simplest way to access settings from Python code:

```
>>> from scrapy.conf import settings
>>> print settings['LOG_ENABLED']
True
```

In other words, settings can be accessed like a dict, but it's usually preferred to extract the setting in the format you need it to avoid type errors. In order to do that you'll have to use one of the following methods:

**class scrapy.conf.Settings**

There is a (singleton) Settings object automatically instantiated when the `scrapy.conf` module is loaded, and it's usually accessed like this:

```
>>> from scrapy.conf import settings
```

**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. For example, both 1 and '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

**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

**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

**getlist** (*name*, *default=None*)

Get a setting value as a list. If the setting original type is a list it will be returned verbatim. 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

## 7.2.4 Rationale for setting names

Setting names are usually prefixed with the component that they configure. For example, proper setting names for a fictional robots.txt extension would be `ROBOTSTXT_ENABLED`, `ROBOTSTXT_OBEY`, `ROBOTSTXT_CACHEDIR`, etc.

## 7.2.5 Built-in settings reference

Here's a list of all available Scrapy settings, in alphabetical order, along with their default values and the scope where they apply.

The scope, where available, shows where the setting is being used, if it's tied to any particular component. In that case the module of that component will be shown, typically an extension, middleware or pipeline. It also means that the component must be enabled in order for the setting to have any effect.

### AWS\_ACCESS\_KEY\_ID

Default: `None`

The AWS access key used by code that requires access to [Amazon Web services](#), such as the *S3 feed storage backend*.

### AWS\_SECRET\_ACCESS\_KEY

Default: `None`

The AWS secret key used by code that requires access to [Amazon Web services](#), such as the *S3 feed storage backend*.

### 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.

### BOT\_VERSION

Default: `1.0`

The version of the bot implemented by this Scrapy project. This will be used to construct the User-Agent by default.



## 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.

## CONCURRENT\_REQUESTS\_PER\_IP

Default: 0

The maximum number of concurrent (ie. simultaneous) requests that will be performed to any single IP. If non-zero, the `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.

## DEFAULT\_ITEM\_CLASS

Default: `'scrapy.item.Item'`

The default class that will be used for instantiating items in the *the Scrapy shell*.

## DEFAULT\_REQUEST\_HEADERS

Default:

```
{
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',
    'Accept-Language': 'en',
}
```

The default headers used for Scrapy HTTP Requests. They're populated in the `DefaultHeadersMiddleware`.

## DEFAULT\_RESPONSE\_ENCODING

Default: `'ascii'`

The default encoding to use for `TextResponse` objects (and subclasses) when no encoding is declared and no encoding could be inferred from the body.

## DEPTH\_LIMIT

Default: 0

The maximum depth that will be allowed to crawl for any site. If zero, no limit will be imposed.

## DEPTH\_PRIORITY

Default: 0

An integer that is used to adjust the request priority based on its depth.

If zero, no priority adjustment is made from depth.

## DEPTH\_STATS

Default: True

Whether to collect maximum depth stats.

## DEPTH\_STATS\_VERBOSE

Default: False

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.

## DOWNLOADER\_DEBUG

Default: False

Whether to enable the Downloader debugging mode.

## DOWNLOADER\_MIDDLEWARES

Default:: {}

A dict containing the downloader middlewares enabled in your project, and their orders. For more info see [Activating a downloader middleware](#).

## DOWNLOADER\_MIDDLEWARES\_BASE

Default:

```
{
    'scrapy.contrib.downloadermiddleware.robotstxt.RobotsTxtMiddleware': 100,
    'scrapy.contrib.downloadermiddleware.httpauth.HttpAuthMiddleware': 300,
    'scrapy.contrib.downloadermiddleware.downloadtimeout.DownloadTimeoutMiddleware': 350,
    'scrapy.contrib.downloadermiddleware.useragent.UserAgentMiddleware': 400,
    'scrapy.contrib.downloadermiddleware.retry.RetryMiddleware': 500,
    'scrapy.contrib.downloadermiddleware.defaultheaders.DefaultHeadersMiddleware': 550,
    'scrapy.contrib.downloadermiddleware.redirect.RedirectMiddleware': 600,
    'scrapy.contrib.downloadermiddleware.cookies.CookiesMiddleware': 700,
    'scrapy.contrib.downloadermiddleware.httpproxy.HttpProxyMiddleware': 750,
    'scrapy.contrib.downloadermiddleware.httpcompression.HttpCompressionMiddleware': 800,
    'scrapy.contrib.downloadermiddleware.chunked.ChunkedTransferMiddleware': 830,
    'scrapy.contrib.downloadermiddleware.stats.DownloaderStats': 850,
    'scrapy.contrib.downloadermiddleware.httppcache.HttpCacheMiddleware': 900,
}
```

A dict containing the downloader middlewares enabled by default in Scrapy. 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 spider. 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`.

You can also change this setting per spider.

## 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',
}
```

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.

## DOWNLOAD\_TIMEOUT

Default: 180

The amount of time (in secs) that the downloader will wait before timing out.

## DUPEFILTER\_CLASS

Default: `'scrapy.dupefilter.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.

## EDITOR

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).

## ENCODING\_ALIASES

Default: `{}`

A mapping of custom encoding aliases for your project, where the keys are the aliases (and must be lower case) and the values are the encodings they map to.

This setting extends the `ENCODING_ALIASES_BASE` setting which contains some default mappings.

## ENCODING\_ALIASES\_BASE

Default:

```
{
    # gb2312 is superseded by gb18030
    'gb2312': 'gb18030',
    'chinese': 'gb18030',
    'cp936': 'gb18030',
    'euc-cn': 'gb18030',
    'euccn': 'gb18030',
    'eucgb2312-cn': 'gb18030',
    'gb2312-1980': 'gb18030',
    'gb2312-80': 'gb18030',
    'iso-ir-58': 'gb18030',
}
```

```

# gbk is superseded by gb18030
'gbk': 'gb18030',
'936': 'gb18030',
'cp936': 'gb18030',
'ms936': 'gb18030',
# latin_1 is a subset of cp1252
'latin_1': 'cp1252',
'iso-8859-1': 'cp1252',
'iso8859-1': 'cp1252',
'8859': 'cp1252',
'cp819': 'cp1252',
'latin': 'cp1252',
'latin1': 'cp1252',
'l1': 'cp1252',
# others
'zh-cn': 'gb18030',
'win-1251': 'cp1251',
'macintosh' : 'mac_roman',
'x-sjis': 'shift_jis',
}

```

The default encoding aliases defined in Scrapy. Don't override this setting in your project, override `ENCODING_ALIASES` instead.

The reason why `ISO-8859-1` (and all its aliases) are mapped to `CP1252` is due to a well known browser hack. For more information see: [Character encodings in HTML](#).

## EXTENSIONS

Default:: {}

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

## EXTENSIONS\_BASE

Default:

```

{
    'scrapy.contrib.corestats.CoreStats': 0,
    'scrapy.webservice.WebService': 0,
    'scrapy.telnet.TelnetConsole': 0,
    'scrapy.contrib.memusage.MemoryUsage': 0,
    'scrapy.contrib.memdebug.MemoryDebugger': 0,
    'scrapy.contrib.clospider.CloseSpider': 0,
    'scrapy.contrib.feedexport.FeedExporter': 0,
    'scrapy.contrib.spidercontext.SpiderContext': 0,
    'scrapy.contrib.logstats.LogStats': 0,
    'scrapy.contrib.spiderstate.SpiderState': 0,
}

```

The list of available extensions. Keep in mind that some of them need to be enabled through a setting. By default, this setting contains all stable built-in extensions.

For more information See the *extensions user guide* and the *list of available extensions*.

## ITEM\_PIPELINES

Default: []

The item pipelines to use (a list of classes).

Example:

```
ITEM_PIPELINES = [
    'mybot.pipeline.validate.ValidateMyItem',
    'mybot.pipeline.validate.StoreMyItem'
]
```

## 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\_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.contrib.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.contrib.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\_NOTIFY\_MAIL

Default: False

Scope: scrapy.contrib.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.contrib.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.contrib.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 and  $1.5 * \text{DOWNLOAD\_DELAY}$ ) while fetching requests from the same spider.

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.

## 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\_MAX\_METAREFRESH\_DELAY

Default: 100

Some sites use meta-refresh for redirecting to a session expired page, so we restrict automatic redirection to a maximum delay (in seconds)

## REDIRECT\_PRIORITY\_ADJUST

Default: +2

Adjust redirect request priority relative to original request. A negative priority adjust means more priority.



## ROBOTSTXT\_OBEY

Default: False

Scope: `scrapy.contrib.downloadermiddleware.robotstxt`

If enabled, Scrapy will respect robots.txt policies. For more information see [RobotsTxtMiddleware](#)

## SCHEDULER

Default: `'scrapy.core.scheduler.Scheduler'`

The scheduler to use for crawling.

## SPIDER\_MIDDLEWARES

Default:: {}

A dict containing the spider middlewares enabled in your project, and their orders. For more info see [Activating a spider middleware](#).

## SPIDER\_MIDDLEWARES\_BASE

Default:

```
{
    'scrapy.contrib.spidermiddleware.httperror.HttpErrorMiddleware': 50,
    'scrapy.contrib.spidermiddleware.offsite.OffsiteMiddleware': 500,
    'scrapy.contrib.spidermiddleware.referrer.RefererMiddleware': 700,
    'scrapy.contrib.spidermiddleware.urllength.UrlLengthMiddleware': 800,
    'scrapy.contrib.spidermiddleware.depth.DepthMiddleware': 900,
}
```

A dict containing the spider middlewares enabled by default in Scrapy. You should never modify this setting in your project, modify `SPIDER_MIDDLEWARES` instead. For more info see [Activating a spider middleware](#).

## SPIDER\_MODULES

Default: []

A list of modules where Scrapy will look for spiders.

Example:

```
SPIDER_MODULES = ['mybot.spiders_prod', 'mybot.spiders_dev']
```

## STATS\_CLASS

Default: `'scrapy.statscol.MemoryStatsCollector'`

The class to use for collecting stats (must implement the Stats Collector API, or subclass the StatsCollector class).

## STATS\_DUMP

Default: `True`

Dump (to the Scrapy log) the *Scrapy stats* collected during the crawl. The spider-specific stats are logged when the spider is closed, while the global stats are dumped when the Scrapy process finishes.

For more info see: *Stats Collection*.

## STATS\_ENABLED

Default: `True`

Enable 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.

## URLLENGTH\_LIMIT

Default: `2083`

Scope: `contrib.spidermiddleware.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>

## USER\_AGENT

Default: `"%s/%s" % (BOT_NAME, BOT_VERSION)`

The default User-Agent to use when crawling, unless overridden.

## 7.3 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.

Finally, for more detailed information about signals internals see the documentation of [pydispatcher](#) (the which the signal dispatching mechanism is based on).

### 7.3.1 Deferred signal handlers

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

### 7.3.2 Built-in signals reference

Here's the list of Scrapy built-in signals and their meaning.

#### **engine\_started**

```
scrapy.signals.engine_started()
```

Sent when the Scrapy engine has started crawling.

This signal supports returning deferreds from their handlers.

---

**Note:** This signal may be fired *after* the [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** ([Item](#) object) – the item scraped
- **response** ([Response](#) object) – the response from where the item was scraped
- **spider** ([BaseSpider](#) object) – the spider which scraped the item

## item\_dropped

`scrapy.signals.item_dropped(item, spider, exception)`

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** (`Item` object) – the item dropped from the *Item Pipeline*
- **spider** (`BaseSpider` object) – the spider which scraped the item
- **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** (`BaseSpider` 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** (`BaseSpider` 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` (`BaseSpider` 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).

#### **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** (`BaseSpider` object) – the spider which raised the exception

### `request_received`

`scrapy.signals.request_received` (*request, spider*)

Sent when the engine receives a `Request` from a spider.

This signal does not support returning deferreds from their handlers.

#### **Parameters**

- **request** (`Request` object) – the request received
- **spider** (`BaseSpider` object) – the spider which generated 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** (`BaseSpider` 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** (`BaseSpider` object) – the spider for which the response is intended

## 7.4 Exceptions

### 7.4.1 Built-in Exceptions reference

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

#### DropItem

**exception** `scrapy.exceptions.DropItem`

The exception that must be raised by item pipeline stages to stop processing an Item. For more information see *Item Pipeline*.

#### CloseSpider

**exception** `scrapy.exceptions.CloseSpider` (*reason='cancelled'*)

This exception can be raised from a spider callback to request the spider to be closed/stopped. Supported arguments:

**Parameters** **reason** (*str*) – the reason for closing

For example:

```
def parse_page(self, response):
    if 'Bandwidth exceeded' in response.body:
        raise CloseSpider('bandwidth_exceeded')
```

#### 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 constructor.

## NotSupported

**exception** scrapy.exceptions.**NotSupported**

This exception is raised to indicate an unsupported feature.

## 7.5 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.

### 7.5.1 Using Item Exporters

If you are in a hurry, and just want to use an Item Exporter to output scraped data see the *Feed exports*. Otherwise, if you want to know how Item Exporters work or need more custom functionality (not covered by the default exports), continue reading below.

In order to use an Item Exporter, you must instantiate it with its required args. Each Item Exporter requires different arguments, so check each exporter documentation to be sure, in *Built-in Item Exporters reference*. After you have instantiated you 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:

```

from scrapy.xlib.pydispatch import dispatcher
from scrapy import signals
from scrapy.contrib.exporter import XmlItemExporter

class XmlExportPipeline(object):

    def __init__(self):
        dispatcher.connect(self.spider_opened, signals.spider_opened)
        dispatcher.connect(self.spider_closed, signals.spider_closed)
        self.files = {}

    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
```

## 7.5.2 Serialization of item fields

By default, the field values are passed unmodified to the underlying serialization library, and the decision of how to serialize them is delegated to each particular serialization library.

However, you can customize how each field value is serialized *before it is passed to the serialization library*.

There are two ways to customize how a field will be serialized, which are described next.

### 1. Declaring a serializer in the field

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:

```
from scrapy.item import Item, Field

def serialize_price(value):
    return '$ %s' % str(value)

class Product(Item):
    name = Field()
    price = Field(serializer=serialize_price)
```

### 2. Overriding the `serialize_field()` method

You can also override the `serialize()` method to customize how your field value will be exported.

Make sure you call the base class `serialize()` method after your custom code.

Example:

```
from scrapy.contrib.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)
```

## 7.5.3 Built-in Item Exporters reference

Here is a list of the Item Exporters bundled with Scrapy. Some of them contain output examples, which assume you're exporting these two items:

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



## BaseItemExporter

```
class scrapy.contrib.exporter.BaseItemExporter (fields_to_export=None,          ex-
                                              port_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) – the field being serialized
- **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.

**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.

**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.contrib.exporter.XmlItemExporter (file,                          item_element='item',
                                              root_element='items', **kwargs)
```

Exports Items in XML format to the specified file object.

### Parameters

- **file** – the file-like object to use for exporting the data.
- **root\_element** (*str*) – The name of root element in the exported XML.
- **item\_element** (*str*) – The name of each item element in the exported XML.

The additional keyword arguments of this constructor are passed to the `BaseItemExporter` constructor.

A typical output of this exporter would be:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item>
    <name>Color TV</name>
    <price>1200</price>
  </item>
  <item>
    <name>DVD player</name>
    <price>200</price>
  </item>
</items>
```

Unless overridden in the `serialize_field()` method, multi-valued fields are exported by serializing each value inside a `<value>` element. This is for convenience, as multi-valued fields are very common.

For example, the item:

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

Would be serialized as:

```
<?xml version="1.0" encoding="utf-8"?>
<items>
  <item>
    <name>
      <value>John</value>
      <value>Doe</value>
    </name>
    <age>23</age>
  </item>
</items>
```

## CsvItemExporter

```
class scrapy.contrib.exporter.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

**class** scrapy.contrib.exporter.**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

**class** scrapy.contrib.exporter.**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

**class** scrapy.contrib.exporter.**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.contrib.exporter.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.

**Command line tool** Learn about the command-line tool and see all *available commands*.

**Requests and Responses** Understand the classes used to represent HTTP requests and responses.

**Settings** Learn how to configure Scrapy and see all *available settings*.

**Signals** See all available signals and how to work with them.

**Exceptions** See all available exceptions and their meaning.

**Item Exporters** Quickly export your scraped items to a file (XML, CSV, etc).

---

## All the rest

---

### 8.1 Contributing to Scrapy

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-developers](#) mailing list and share your ideas on how to improve Scrapy. We're always open to suggestions.

#### 8.1.1 Reporting bugs

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.
- 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.

#### 8.1.2 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.

### 8.1.3 Submitting patches

The best way to submit a patch is to issue a [pull request](#) on Github, optionally creating a new issue first.

Alternatively, we also accept the patches in the traditional way of sending them to the [scrapy-developers](#) list.

Regardless of which mechanism you use, 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) in [scrapy-developers](#) first, 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.

### 8.1.4 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.

### 8.1.5 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.

### 8.1.6 Tests

Tests are implemented using the [Twisted unit-testing framework](#) called `trial`.

#### Running tests

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

```
bin/runtests.sh (on unix)
bin\runtests.bat (on windows)
```

To run a specific test (say `scrapy.tests.test_contrib_loader`) use:

```
bin/runtests.sh scrapy.tests.test_contrib_loader (on unix)
bin\runtests.bat scrapy.tests.test_contrib_loader (on windows)
```

## 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 `scrapy.tests` package (`scrapy/tests` directory). Their module name typically resembles the full path of the module they're testing. For example, the item loaders code is in:

```
scrapy.contrib.loader
```

And their unit-tests are in:

```
scrapy.tests.test_contrib_loader
```

## 8.2 Versioning and API Stability

### 8.2.1 Versioning

Scrapy uses the [odd-numbered versions for development releases](#).

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. So far, only zero is available for *A* as Scrapy hasn't yet reached 1.0.
- *B* is the release number. This will include many changes including features and things that possibly break backwards compatibility. Even *B*s will be stable branches, and odd *B*s will be development.
- *C* is the bugfix release number.

For example:

- *0.14.1* is the first bugfix release of the *0.14* series (safe to use in production)

### 8.2.2 API Stability

API stability is one of Scrapy major goals for the *1.0* release, which doesn't have a due date scheduled yet.

Methods or functions that start with a single dash (`_`) are private and should never be relied as stable. Besides those, the plan is to stabilize and document the entire API, as we approach the 1.0 release.

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.

## 8.3 Experimental features

This section documents experimental Scrapy features that may become stable in future releases, but whose API is not yet stable. Use them with caution, and subscribe to the [mailing lists](#) to get notified of any changes.

Since it's not revised so frequently, this section may contain documentation which is outdated, incomplete or overlapping with stable documentation (until it's properly merged) . Use at your own risk.

**Warning:** This documentation is a work in progress. Use at your own risk.

### 8.3.1 DjangoItem

DjangoItem is a class of item that gets its fields definition from a Django model, you simply create a DjangoItem and specify what Django model it relates to.

Besides of getting the model fields defined on your item, DjangoItem provides a method to create and populate a Django model instance with the item data.

#### Using DjangoItem

DjangoItem works much like ModelForms in Django, you create a subclass and define its django\_model attribute to be a valid Django model. With this you will get an item with a field for each Django model field.

In addition, you can define fields that aren't present in the model and even override fields that are present in the model defining them in the item.

Let's see some examples:

Django model for the examples:

```
class Person(models.Model):
    name = models.CharField(max_length=255)
    age = models.IntegerField()
```

Defining a basic DjangoItem:

```
class PersonItem(DjangoItem):
    django_model = Person
```

DjangoItem work just like Item:

```
p = PersonItem()
p['name'] = 'John'
p['age'] = '22'
```

To obtain the Django model from the item, we call the extra method save() of the DjangoItem:

```
>>> person = p.save()
>>> person.name
'John'
>>> person.age
'22'
>>> person.id
1
```

As you see the model is already saved when we call save(), we can prevent this by calling it with commit=False. We can use commit=False in save() method to obtain an unsaved model:

```
>>> person = p.save(commit=False)
>>> person.name
'John'
>>> person.age
```



```
'22'  
>>> person.id  
None
```

As said before, we can add other fields to the item:

```
class PersonItem(DjangoItem):  
    django_model = Person  
    sex = Field()  
  
p = PersonItem()  
p['name'] = 'John'  
p['age'] = '22'  
p['sex'] = 'M'
```

---

**Note:** fields added to the item won't be taken into account when doing a `save()`

---

And we can override the fields of the model with your own:

```
class PersonItem(DjangoItem):  
    django_model = Person  
    name = Field(default='No Name')
```

This is useful to provide properties to the field, like a default or any other property that your project uses.

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

***Versioning and API Stability*** Understand Scrapy versioning and API stability.

***Experimental features*** Learn about bleeding-edge features.



**S**

`scrapy.conf`, 138  
`scrapy.contrib.closespider`, 128  
`scrapy.contrib.corestats`, 127  
`scrapy.contrib.debug`, 128  
`scrapy.contrib.downloadermiddleware`, 114  
`scrapy.contrib.downloadermiddleware.chunked`, 119  
`scrapy.contrib.downloadermiddleware.cookies`, 116  
`scrapy.contrib.downloadermiddleware.defaultheaders`, 116  
`scrapy.contrib.downloadermiddleware.downloadtimeout`, 116  
`scrapy.contrib.downloadermiddleware.httpauth`, 116  
`scrapy.contrib.downloadermiddleware.httpcache`, 117  
`scrapy.contrib.downloadermiddleware.httpcompression`, 119  
`scrapy.contrib.downloadermiddleware.httpproxy`, 119  
`scrapy.contrib.downloadermiddleware.redirect`, 119  
`scrapy.contrib.downloadermiddleware.retry`, 120  
`scrapy.contrib.downloadermiddleware.robotstxt`, 120  
`scrapy.contrib.downloadermiddleware.stats`, 121  
`scrapy.contrib.downloadermiddleware.useragent`, 121  
`scrapy.contrib.exporter`, 155  
`scrapy.contrib.linkextractors`, 62  
`scrapy.contrib.linkextractors.sgml`, 62  
`scrapy.contrib.loader`, 45  
`scrapy.contrib.loader.processor`, 51  
`scrapy.contrib.logstats`, 126  
`scrapy.contrib.memdebug`, 127  
`scrapy.contrib.memusage`, 127  
`scrapy.contrib.pipeline.images`, 97  
`scrapy.contrib.spidermiddleware`, 122  
`scrapy.contrib.spidermiddleware.depth`, 123  
`scrapy.contrib.spidermiddleware.httperror`, 123  
`scrapy.contrib.spidermiddleware.offsite`, 123  
`scrapy.contrib.spidermiddleware.referrer`, 124  
`scrapy.contrib.spidermiddleware.urllength`, 124  
`scrapy.contrib.spiders`, 32  
`scrapy.contrib.statsmailer`, 128  
`scrapy.contrib.webservice`, 75  
`scrapy.contrib.webservice.crawler`, 75  
`scrapy.contrib.webservice.enginestatus`, 76  
`scrapy.contrib.webservice.stats`, 75  
`scrapy.contrib_exp.djangoitem`, 164  
`scrapy.exceptions`, 154  
`scrapy.extension`, 126  
`scrapy.http`, 131  
`scrapy.item`, 26  
`scrapy.log`, 66  
`scrapy.mail`, 70  
`scrapy.selector`, 42  
`scrapy.signals`, 151  
`scrapy.spider`, 30  
`scrapy.statscol`, 68  
`scrapy.telnet`, 72  
`scrapy.utils.trackref`, 92  
`scrapy.webservice`, 127