Scrapy

tutorialspoint

SIMPLY EASY LEARNING

www.tutorialspoint.com

https://www.facebook.com/tutorialspointindia

https://twitter.com/tutorialspoint

About the Tutorial

Scrapy is a fast, open-source web crawling framework written in Python, used to extract the data from the web page with the help of selectors based on XPath.

Audience

This tutorial is designed for software programmers who need to learn Scrapy web crawler from scratch.

Prerequisites

You should have a basic understanding of Computer Programming terminologies and Python. A basic understanding of XPath is a plus.

Copyright & Disclaimer

Table of Contents

Scrapy Basic Concepts

1. Scrapy – Overview

Scrapy is a fast, open-source web crawling framework written in Python, used to extract the data from the web page with the help of selectors based on XPath.

Scrapy was first released on June 26, 2008 licensed under BSD, with a milestone 1.0 releasing in June 2015.

Why Use Scrapy?

- It is easier to build and scale large crawling projects.
- It has a built-in mechanism called Selectors, for extracting the data from websites.
- It handles the requests asynchronously and it is fast.
- It automatically adjusts crawling speed using Auto-throttling mechanism.
- Ensures developer accessibility.

Features of Scrapy

- Scrapy is an open source and free to use web crawling framework.
- Scrapy generates feed exports in formats such as JSON, CSV, and XML.
- Scrapy has built-in support for selecting and extracting data from sources either by XPath or CSS expressions.
- Scrapy based on crawler, allows extracting data from the web pages automatically.

Advantages

- Scrapy is easily extensible, fast, and powerful.
- It is a cross-platform application framework (Windows, Linux, Mac OS and BSD).
- Scrapy requests are scheduled and processed asynchronously.
- Scrapy comes with built-in service called **Scrapyd** which allows to upload projects and control spiders using JSON web service.
- It is possible to scrap any website, though that website does not have API for raw data access.

Disadvantages

- Scrapy is only for Python 2.7. +
- Installation is different for different operating systems.

2. Scrapy – Environment

In this chapter, we will discuss how to install and set up Scrapy. Scrapy must be installed with Python.

Scrapy can be installed by using **pip**. To install, run the following command:

```
pip install Scrapy
```

Windows

Note: Python 3 is not supported on Windows OS.

Step 1: Install Python 2.7 from Python

Set environmental variables by adding the following paths to the PATH:

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

You can check the Python version using the following command:

```
python --version
```

Step 2: Install OpenSSL.

Add C:\OpenSSL-Win32\bin in your environmental variables.

Note: OpenSSL comes preinstalled in all operating systems except Windows.

Step 3: Install Visual C++ 2008 redistributables.

Step 4: Install pywin32.

Step 5: Install pip for Python versions older than 2.7.9.

You can check the pip version using the following command:

```
pip --version
```

Step 6: To install scrapy, run the following command:

```
pip install Scrapy
```

Anaconda

If you have anaconda or miniconda installed on your machine, run the following command to install Scrapy using conda:

```
conda install -c scrapinghub scrapy
```

Scrapinghub company supports official conda packages for Linux, Windows, and OS X.

Note: It is recommended to install Scrapy using the above command if you have issues installing via pip.

Ubuntu 9.10 or Above

The latest version of Python is pre-installed on Ubuntu OS. Use the Ubuntu packages apt-gettable provided by Scrapinghub. To use the packages:

Step 1: You need to import the GPG key used to sign Scrapy packages into APT keyring:

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv 627220E7
```

Step 2: Next, use the following command to create /etc/apt/sources.list.d/scrapy.list file:

```
echo 'deb http://archive.scrapy.org/ubuntu scrapy main' | sudo tee
/etc/apt/sources.list.d/scrapy.list
```

Step 3: Update package list and install scrapy:

```
sudo apt-get update && sudo apt-get install scrapy
```

Archlinux

You can install Scrapy from *AUR Scrapy package* using the following command:

```
yaourt -S scrapy
```

Mac OS X

Use the following command to install Xcode command line tools:

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

Instead of using system Python, install a new updated version that doesn't conflict with the rest of your system.

Step 1: Install homebrew.

Step 2: Set environmental PATH variable to specify that homebrew packages should be used before system packages:

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

Step 3: To make sure the changes are done, reload **.bashrc** using the following command:

```
source ~/.bashrc
```

Step 4: Next, install Python using the following command:

```
brew install python
```

Step 5: Install Scrapy using the following command:

```
pip install Scrapy
```

3. Scrapy – Command Line Tools

Description

The Scrapy command line tool is used for controlling Scrapy, which is often referred to as **'Scrapy tool'**. It includes the commands for various objects with a group of arguments and options.

Configuration Settings

Scrapy will find configuration settings in the **scrapy.cfg** file. Following are a few locations:

- C:\scrapy(project folder)\scrapy.cfg in the system
- ~/.config/scrapy.cfg ($XDG_CONFIG_HOME) and ~/.scrapy.cfg ($HOME) for global settings
- You can find the scrapy.cfg inside the root of the project.

Scrapy can also be configured using the following environment variables:

- SCRAPY_SETTINGS_MODULE
- SCRAPY_PROJECT
- SCRAPY_PYTHON_SHELL

Default Structure Scrapy Project

The following structure shows the default file structure of the Scrapy project.

```
scrapy.cfg                - Deploy the configuration file
project_name/             - Name of the project
   _init_.py
   items.py               - It is project's items file
   pipelines.py           - It is project's pipelines file
   settings.py            - It is project's settings file
   spiders                - It is the spiders directory
      _init_.py
      spider_name.py
    . . .
```

The **scrapy.cfg** file is a project root directory, which includes the project name with the project settings. For instance:

```
[settings]
default = [name of the project].settings
```

```
[deploy]
#url = http://localhost:6800/
project = [name of the project]
```

Using Scrapy Tool

Scrapy tool provides some usage and available commands as follows:

```
Scrapy X.Y  - no active project
Usage:
   scrapy  [options] [arguments]
Available commands:
   crawl      It puts spider (handle the URL) to work for crawling data
   fetch      It fetches the response from the given URL
```

Creating a Project

You can use the following command to create the project in Scrapy:

```
scrapy startproject project_name
```

This will create the project called **project_name** directory. Next, go to the newly created project, using the following command:

```
cd  project_name
```

Controlling Projects

You can control the project and manage them using the Scrapy tool and also create the new spider, using the following command:

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

The commands such as crawl, etc. must be used inside the Scrapy project. You will come to know which commands must run inside the Scrapy project in the coming section.

Scrapy contains some built-in commands, which can be used for your project. To see the list of available commands, use the following command:

```
scrapy -h
```

When you run the following command, Scrapy will display the list of available commands as listed:

- **fetch**: It fetches the URL using Scrapy downloader.

- **runspider**: It is used to run self-contained spider without creating a project.
- **settings**: It specifies the project setting value.
- **shell**: It is an interactive scraping module for the given URL.
- **startproject**: It creates a new Scrapy project.
- **version**: It displays the Scrapy version.
- **view**: It fetches the URL using Scrapy downloader and show the contents in a browser.

You can have some project related commands as listed:

- **crawl**: It is used to crawl data using the spider.
- **check**: It checks the items returned by the crawled command.
- **list**: It displays the list of available spiders present in the project.
- **edit**: You can edit the spiders by using the editor.
- **parse**: It parses the given URL with the spider.
- **bench**: It is used to run quick benchmark test (Benchmark tells how many number of pages can be crawled per minute by Scrapy).

Custom Project Commands

You can build a custom project command with **COMMANDS_MODULE** setting in Scrapy project. It includes a default empty string in the setting. You can add the following custom command:

```
COMMANDS_MODULE = 'mycmd.commands'
```

Scrapy commands can be added using the scrapy.commands section in the setup.py file shown as follows:

```
from setuptools import setup, find_packages


setup(name='scrapy-module_demo',
  entry_points={
    'scrapy.commands': [
      'cmd_demo=my_module.commands:CmdDemo',
    ],
  },
)
```

The above code adds **cmd_demo** command in the **setup.py** file.

4. Scrapy – Spiders

Description

Spider is a class responsible for defining how to follow the links through a website and extract the information from the pages.

The default spiders of Scrapy are as follows:

scrapy.Spider

It is a spider from which every other spiders must inherit. It has the following class:

```
class scrapy.spiders.Spider
```

The following table shows the fields of scrapy.Spider class:

Sr. No.	Field & Description
1	**name** It is the name of your spider.
2	**allowed_domains** It is a list of domains on which the spider crawls.
3	**start_urls** It is a list of URLs, which will be the roots for later crawls, where the spider will begin to crawl from.
4	**custom_settings** These are the settings, when running the spider, will be overridden from project wide configuration.
5	**crawler** It is an attribute that links to Crawler object to which the spider instance is bound.
6	**settings** These are the settings for running a spider.
7	**logger** It is a Python logger used to send log messages.

8	**from_crawler(crawler,*args,**kwargs)** It is a class method, which creates your spider. The parameters are: • **crawler**: A crawler to which the spider instance will be bound. • **args(list)**: These arguments are passed to the method *_init_()*. • **kwargs(dict)**: These keyword arguments are passed to the method *_init_()*.
9	**start_requests()** When no particular URLs are specified and the spider is opened for scrapping, Scrapy calls *start_requests()* method.
10	**make_requests_from_url(url)** It is a method used to convert urls to requests.
11	**parse(response)** This method processes the response and returns scrapped data following more URLs.
12	**log(message[,level,component])** It is a method that sends a log message through spiders logger.
13	**closed(reason)** This method is called when the spider closes.

Spider Arguments

Spider arguments are used to specify start URLs and are passed using crawl command with **-a** option, shown as follows:

```
scrapy crawl first_scrapy -a group = accessories
```

The following code demonstrates how a spider receives arguments:

```
import scrapy
class FirstSpider(scrapy.Spider):
    name = "first"
    def __init__(self, group=None, *args, **kwargs):
        super(FirstSpider, self).__init__(*args, **kwargs)
        self.start_urls = ["http://www.example.com/group/%s" % group]
```

Generic Spiders

You can use generic spiders to subclass your spiders from. Their aim is to follow all links on the website based on certain rules to extract data from all pages.

For the examples used in the following spiders, let's assume we have a project with the following fields:

```
import scrapy
from scrapy.item import Item, Field


class First_scrapyItem(scrapy.Item):
    product_title = Field()
    product_link = Field()
    product_description = Field()
```

CrawlSpider

CrawlSpider defines a set of rules to follow the links and scrap more than one page. It has the following class:

```
class scrapy.spiders.CrawlSpider
```

Following are the attributes of CrawlSpider class:

rules

It is a list of rule objects that defines how the crawler follows the link.

The following table shows the rules of CrawlSpider class:

Sr. No.	Rule & Description
1	**LinkExtractor** It specifies how spider follows the links and extracts the data.
2	**callback** It is to be called after each page is scraped.
3	**follow** It specifies whether to continue following links or not.

parse_start_url(response)

It returns either item or request object by allowing to parse initial responses.

Note: Make sure you rename parse function other than parse while writing the rules because the parse function is used by CrawlSpider to implement its logic.

Let’s take a look at the following example, where spider starts crawling demoexample.com's home page, collecting all pages, links, and parses with the *parse_items* method:

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


class DemoSpider(CrawlSpider):
    name = "demo"
    allowed_domains = ["www.demoexample.com"]
    start_urls = ["http://www.demoexample.com"]


    rules = (
    Rule(LinkExtractor(allow =(), restrict_xpaths = ("//div[@class =
'next']",)), callback = "parse_item", follow = True),
    )


    def parse_item(self, response):
        item = DemoItem()
        item["product_title"] = response.xpath("a/text()").extract()
        item["product_link"] = response.xpath("a/@href").extract()
        item["product_description"]  =
response.xpath("div[@class='desc']/text()").extract()
        return items
```

XMLFeedSpider

It is the base class for spiders that scrape from XML feeds and iterates over nodes. It has the following class:

```
class scrapy.spiders.XMLFeedSpider
```

The following table shows the class attributes used to set an iterator and a tag name:

Sr. No.	Attribute & Description
1	**iterator** It defines the iterator to be used. It can be either *iternodes*, *html* or *xml*. Default is *iternodes*.

2	**itertag** It is a string with node name to iterate.
3	**namespaces** It is defined by list of (prefix, uri) tuples that automatically registers namespaces using *register_namespace()* method.
4	**adapt_response(response)** It receives the response and modifies the response body as soon as it arrives from spider middleware, before spider starts parsing it.
5	**parse_node(response,selector)** It receives the response and a selector when called for each node matching the provided tag name. **Note**: Your spider won't work if you don't override this method.
6	**process_results(response,results)** It returns a list of results and response returned by the spider.

CSVFeedSpider

It iterates through each of its rows, receives a CSV file as a response, and calls *parse_row()* method. It has the following class:

```
class scrapy.spiders.CSVFeedSpider
```

The following table shows the options that can be set regarding the CSV file:

Sr. No.	Option & Description
1	**delimiter** It is a string containing a comma(',') separator for each field.
2	**quotechar** It is a string containing quotation mark('"') for each field.
3	**headers** It is a list of statements from where the fields can be extracted.
4	**parse_row(response,row)** It receives a response and each row along with a key for header.

CSVFeedSpider Example

```
from scrapy.spiders import CSVFeedSpider
from demoproject.items import DemoItem

class DemoSpider(CSVFeedSpider):
    name = "demo"
    allowed_domains = ["www.demoexample.com"]
    start_urls = ["http://www.demoexample.com/feed.csv"]
    delimiter = ";"
    quotechar = "'"
    headers = ["product_title", "product_link", "product_description"]

    def parse_row(self, response, row):
        self.logger.info("This is row: %r", row)

        item = DemoItem()
        item["product_title"] = row["product_title"]
        item["product_link"] = row["product_link"]
        item["product_description"] = row["product_description"]
        return item
```

SitemapSpider

SitemapSpider with the help of Sitemaps crawl a website by locating the URLs from robots.txt. It has the following class:

```
class scrapy.spiders.SitemapSpider
```

The following table shows the fields of SitemapSpider:

Sr. No.	Field & Description
1	**sitemap_urls** A list of URLs which you want to crawl pointing to the sitemaps.
2	**sitemap_rules** It is a list of tuples (regex, callback), where regex is a regular expression, and callback is used to process URLs matching a regular expression.

3	**sitemap_follow** It is a list of sitemap's regexes to follow.
4	**sitemap_alternate_links** Specifies alternate links to be followed for a single url.

SitemapSpider Example

The following SitemapSpider processes all the URLs:

```
from scrapy.spiders import SitemapSpider


class DemoSpider(SitemapSpider):
    urls = ["http://www.demoexample.com/sitemap.xml"]


    def parse(self, response):
        # You can scrap items here
```

The following SitemapSpider processes some URLs with callback:

```
from scrapy.spiders import SitemapSpider


class DemoSpider(SitemapSpider):
    urls = ["http://www.demoexample.com/sitemap.xml"]
    rules = [
        ("/item/", "parse_item"),
        ("/group/", "parse_group"),
    ]


    def parse_item(self, response):
        # you can scrap item here


    def parse_group(self, response):
        # you can scrap group here
```

The following code shows sitemaps in the robots.txt whose url has **/sitemap_company**:

```
from scrapy.spiders import SitemapSpider

class DemoSpider(SitemapSpider):
    urls = ["http://www.demoexample.com/robots.txt"]
    rules = [
        ("/company/", "parse_company"),
    ]
    sitemap_follow = ["/sitemap_company"]

    def parse_company(self, response):
        # you can scrap company here
```

You can even combine SitemapSpider with other URLs as shown in the following command.

```
from scrapy.spiders import SitemapSpider

class DemoSpider(SitemapSpider):
    urls = ["http://www.demoexample.com/robots.txt"]
    rules = [
        ("/company/", "parse_company"),
    ]

    other_urls = ["http://www.demoexample.com/contact-us"]
    def start_requests(self):
        requests = list(super(DemoSpider, self).start_requests())
        requests += [scrapy.Request(x, self.parse_other) for x in self.other_urls]
        return requests

    def parse_company(self, response):
        # you can scrap company here...

    def parse_other(self, response):
        # you can scrap other here...
```

5. Scrapy – Selectors

Description

When you are scraping the web pages, you need to extract a certain part of the HTML source by using the mechanism called **selectors**, achieved by using either XPath or CSS expressions. Selectors are built upon the **lxml** library, which processes the XML and HTML in Python language.

Use the following code snippet to define different concepts of selectors:

```
<html>
 <head>
  <title>My Website</title>
 </head>
 <body>
  <span>Hello world!!!</span>
  <div class='links'>
   <a href='one.html'>Link 1<img src='image1.jpg'/></a>
   <a href='two.html'>Link 2<img src='image2.jpg'/></a>
   <a href='three.html'>Link 3<img src='image3.jpg'/></a>
  </div>
 </body>
</html>
```

Constructing Selectors

You can construct the selector class instances by passing the **text** or **TextResponse** object. Based on the provided input type, the selector chooses the following rules:

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

Using the above code, you can construct from the text as:

```
Selector(text=body).xpath('//span/text()').extract()
```

It will display the result as:

```
[u'Hello world!!!']
```

You can construct from the response as:

```
response = HtmlResponse(url='http://mysite.com', body=body)
Selector(response=response).xpath('//span/text()').extract()
```

It will display the result as:

```
[u'Hello world!!!']
```

Using Selectors

Using the above simple code snippet, you can construct the XPath for selecting the text which is defined in the title tag as shown below:

```
>>response.selector.xpath('//title/text()')
```

Now, you can extract the textual data using the **.extract()** method shown as follows:

```
>>response.xpath('//title/text()').extract()
```

It will produce the result as:

```
[u'My Website']
```

You can display the name of all elements shown as follows:

```
>>response.xpath('//div[@class="links"]/a/text()').extract()
```

It will display the elements as:

```
Link 1
Link 2
Link 3
```

If you want to extract the first element, then use the method **.extract_first(),** shown as follows:

```
>>response.xpath('//div[@class="links"]/a/text()').extract_first()
```

It will display the element as:

```
Link 1
```

Nesting Selectors

Using the above code, you can nest the selectors to display the page link and image source using the **.xpath()** method, shown as follows:

```
links = response.xpath('//a[contains(@href, "image")]')
for index, link in enumerate(links):
    args = (index, link.xpath('@href').extract(),
link.xpath('img/@src').extract())
    print 'The link %d pointing to url %s and image %s' % args
```

It will display the result as:

```
Link 1 pointing to url [u'one.html'] and image [u'image1.jpg']
Link 2 pointing to url [u'two.html'] and image [u'image2.jpg']
Link 3 pointing to url [u'three.html'] and image [u'image3.jpg']
```

Selectors Using Regular Expressions

Scrapy allows to extract the data using regular expressions, which uses the **.re()** method. From the above HTML code, we will extract the image names shown as follows:

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

The above line displays the image names as:

```
[u'Link 1',
 u'Link 2',
 u'Link 3']
```

Using Relative XPaths

When you are working with XPaths, which starts with the **/**, nested selectors and XPath are related to absolute path of the document, and not the relative path of the selector.

If you want to extract the **<p>** elements, then first gain all div elements:

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

Next, you can extract all the '**p**' elements inside, by prefixing the XPath with a dot as **.//p** as shown below:

```
>>for p in mydiv.xpath('.//p').extract()
```

Using EXSLT Extensions

The EXSLT is a community that issues the extensions to the XSLT (Extensible Stylesheet Language Transformations) which converts XML documents to XHTML documents. You can use the EXSLT extensions with the registered namespace in the XPath expressions as shown in the following table:

Sr. No.	Prefix & Usage	Namespace
1	**re** regular expressions	**http://exslt.org/regular-expressions**
2	**set** set manipulation	**http://exslt.org/sets**

You can check the simple code format for extracting data using regular expressions in the previous section.

There are some XPath tips, which are useful when using XPath with Scrapy selectors. For more information, click this link.

XPath Tips

Using Text Nodes in a Condition

When you are using text nodes in a XPath string function, then use **. (dot)** instead of using **.//text()**, because this produces the collection of text elements called as **node-set**.

For instance:

```
from scrapy import Selector
val = Selector(text='<a href="#">More Info<strong>click here</strong></a>')
```

If you are converting a node-set to a string, then use the following format:

```
>>val.xpath('//a//text()').extract()
```

It will display the element as:

```
[u'More Info',u'click here']
```

and

```
>>val.xpath("string('//a[1]//text())").extract()
```

It results the element as:

```
[u'More Info']
```

Difference Between //node[1] and (//node)[1]

The **//node[1]** displays all the first elements defined under respective parents. The **(//node)[1]** displays only first element in the document.

For instance:

```
from scrapy import Selector
val = Selector(text="""
   <ul class="list">
      <li>one</li>
      <li>one</li>
      <li>one</li>
   </ul>
   <ul class="list">
      <li>four</li>
      <li>five</li>
      <li>six</li>
   </ul>""")
res = lambda x: val.xpath(x).extract()
```

The following line displays all the first **li** elements defined under their respective parents:

```
>>res("//li[1]")
```

It will display the result as

```
[u'<li>one</li>', u'<li>four</li>']
```

You can get the first **li** element of the complete document shown as follows:

```
>>res("(//li)[1]")
```

It will display the result as

```
[u'<li>one</li>']
```

You can also display all the first **li** elements defined under **ul** parent:

```
>>res("//ul//li[1]")
```

It will display the result as

```
[u'<li>one</li>', u'<li>four</li>']
```

You can get the first **li** element defined under **ul** parent in the whole document shown as follows:

```
>>res("(//ul//li)[1]")
```

It will display the result as

```
[u'<li>one</li>']
```

Built-in Selectors Reference

The built-in selectors include the following class:

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

The above class contains the following parameters:

- **response**: It is a HTMLResponse and XMLResponse that selects and extracts the data.
- **text**: It encodes all the characters using the UTF-8 character encoding, when there is no response available.
- **type**: It specifies the different selector types, such as html for HTML Response, xml for XMLResponse type and none for default type. It selects the type depending on the response type or sets to html by default, if it is used with the text.

The built-in selectors contain the following methods:

Sr. No.	Method & Description
1	**xpath(query)** It matches the nodes according to the xpath query and provides the results as SelectorList instance. The parameter *query* specifies the XPATH query to be used.
2	**css(query)** It supplies the CSS selector and gives back the SelectorList instance. The parameter query specifies CSS selector to be used.
3	**extract()** It brings out all the matching nodes as a list of unicode strings.
4	**re(regex)** It supplies the regular expression and brings out the matching nodes as a list of unicode strings. The parameter regex can be used as a regular expression or string, which compiles to regular expression using the re.compile(regex) method.
5	**register_namespace(prefix, uri)** It specifies the namespace used in the selector. You cannot extract the data without registering the namespace from the non-standard namespace.

6	**remove_namespaces()** It discards the namespace and gives permission to traverse the document using the namespace-less xpaths.
7	**__nonzero__()** If the content is selected, then this method returns true, otherwise returns false.

SelectorList Objects

```
class scrapy.selector.SelectorList
```

The SelectorList objects contains the following methods:

Sr. No.	Method & Description
1	**xpath(query)** It uses the .xpath() method for the elements and provides the results as SelectorList instance. The parameter query specifies the arguments as defined in the Selector.xpath() method.
2	**css(query)** It uses the .css() method for the elements and gives back the results as SelectorList instance. The parameter query specifies the arguments as defined in the Selector.css() method.
3	**extract()** It brings out all the elements of the list using the .extract() method and returns the result as a list of unicode strings.
4	**re()** It uses the .re() method for the elements and brings out the elements as a list of unicode strings.
5	**__nonzero__()** If the list is not empty, then this method returns true, otherwise returns false.

The SelectorList objects contain some of the concepts as explained in this link.

SelectorList Objects

Selector Examples on HTML Response

Following are some of the examples on HTMLResponse and we will have HTMLResponse object, which is instantiated with the selector, shown as follows:

```
res = Selector(html_response)
```

You can select the **h2** elements from HTML response body, which returns the SelectorList object as:

```
>>res.xpath("//h2")
```

You can select the **h2** elements from HTML response body, which returns the list of unicode strings as:

```
>>res.xpath("//h2").extract()
```

It returns the h2 elements.

and

```
>>res.xpath("//h2/text()").extract()
```

It returns the text defined under h2 tag and does not include h2 tag elements.

You can run through the **p** tags and display the class attribute as:

```
for ele in res.xpath("//p"):
    print ele.xpath("@class").extract()
```

Selector Examples on XML Response

Following are some of the examples on XMLResponse and we will have XMLResponse object, which is instantiated with the selector, shown as follows:

```
res = Selector(xml_response)
```

You can select the description elements from XML response body, which returns the SelectorList object as:

```
>>res.xpath("//description")
```

You can get the price value from the Google Base XML feed by registering a namespace as:

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

Removing Namespaces

When you are creating the Scrapy projects, you can remove the namespaces using the Selector.remove_namespaces() method and use the element names to work appropriately with XPaths.

There are two reasons for not calling the namespace removal procedure always in the project:

- You can remove the namespace which requires repeating the document and modifying the all elements that leads to expensive operation to crawl documents by Scrapy.

- In some cases, you need to use namespaces and these may conflict with the some element names and namespaces. This type of case occurs very often.

6. Scrapy – Items

Description

Scrapy process can be used to extract the data from sources such as web pages using the spiders. Scrapy uses **Item** class to produce the output whose objects are used to gather the scraped data.

Declaring Items

You can declare the items using the class definition syntax along with the field objects shown as follows:

```
import scrapy
class MyProducts(scrapy.Item):
    productName = Field()
    productLink = Field()
    imageURL = Field()
    price = Field()
    size = Field()
```

Item Fields

The item fields are used to display the metadata for each field. As there is no limitation of values on the field objects, the accessible metadata keys does not ontain any reference list of the metadata. The field objects are used to specify all the field metadata and you can specify any other field key as per your requirement in the project. The field objects can be accessed using the Item.fields attribute.

Working with Items

There are some common functions which can be defined when you are working with the items. For more information, click this link.

Items

Creating Items

You can create the items as shown in the following format:

```
>>myproduct =  Product(name='Mouse', price=400)
>>print myproduct
```

The above code produces the following result:

```
Product(name='Mouse', price=400)
```

Getting Field Values

You can get the field values as shown in the following way:

```
>>myproduct[name]
```

It will print result as 'Mouse'

Or in another way, you can get the value using **get()** method as:

```
>>myproduct.get(name)
```

It will print result as 'Mouse'

You can also check whether the field is present or not using the following way:

```
>>'name' in myproduct
```

It will print the result as 'True'

Or

```
>>'fname' in myproduct
```

It will print the result as 'False'

Setting Field Values

You can set value for the field shown as follows:

```
>>myproduct['fname'] = 'smith'

>>myproduct['fname']
```

Accessing all Populated Values

It is possible to access all the values, which reside in the 'Product' item.

```
>>myproduct.keys()
```

It will print the result as:

```
['name', 'price']
```

Or you can access all the values along with the field values shown as follows:

```
>>myproduct.items()
```

It will print the result as:

```
[('name', 'Mouse'), ('price', 400)]
```

It's possible to copy items from one field object to another field object as described:

```
>> myresult = Product(myproduct)
>> print myresult
```

It will print the output as:

```
Product(name='Mouse', price=400)
>> myresult1 = myresult.copy()
>> print myresult1
```

It will print the output as:

```
Product(name='Mouse', price=400)
```

Extending Items

The items can be extended by stating the subclass of the original item. For instance:

```
class MyProductDetails(Product):
   original_rate = scrapy.Field(serializer=str)
   discount_rate = scrapy.Field()
```

You can use the existing field metadata to extend the field metadata by adding more values or changing the existing values as shown in the following code:

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

Item Objects

The item objects can be specified using the following class which provides the new initialized item from the given argument:

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

The Item provides a copy of the constructor and provides an extra attribute that is given by the items in the fields.

Field Objects

The field objects can be specified using the following class in which the Field class doesn't issue the additional process or attributes:

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

7. Scrapy – Item Loaders

Description

Item loaders provide a convenient way to fill the items that are scraped from the websites.

Declaring Item Loaders

The declaration of Item Loaders is like Items.

For example:

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

class DemoLoader(ItemLoader):

    default_output_processor = TakeFirst()

    title_in = MapCompose(unicode.title)
    title_out = Join()

    size_in = MapCompose(unicode.strip)

    # you can continue scraping here
```

In the above code, you can see that input processors are declared using **_in** suffix and output processors are declared using **_out** suffix.

The **ItemLoader.default_input_processor** and **ItemLoader.default_output_processor** attributes are used to declare default input/output processors.

Using Item Loaders to Populate Items

To use Item Loader, first instantiate with dict-like object or without one where the item uses Item class specified in **ItemLoader.default_item_class** attribute.

- You can use selectors to collect values into the Item Loader.
- You can add more values in the same item field, where Item Loader will use an appropriate handler to add these values.

The following code demonstrates how items are populated using Item Loaders:

```
from scrapy.loader import ItemLoader
from demoproject.items import Demo

def parse(self, response):
    l = ItemLoader(item = Product(), response = response)
    l.add_xpath("title", "//div[@class='product_title']")
    l.add_xpath("title", "//div[@class='product_name']")
    l.add_xpath("desc", "//div[@class='desc']")
    l.add_css("size", "div#size]")
    l.add_value("last_updated", "yesterday")
    return l.load_item()
```

As shown above, there are two different *XPaths* from which the **title** field is extracted using **add_xpath()** method:

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

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

Thereafter, a similar request is used for **desc** field. The **size** data is extracted using **add_css()** method and **last_updated** is filled with a value "yesterday" using **add_value()** method.

Once all the data is collected, call **ItemLoader.load_item()** method which returns the items filled with data extracted using **add_xpath()**, **add_css()** and **add_value()** methods.

Input and Output Processors

Each field of an Item Loader contains one input processor and one output processor.

- When data is extracted, input processor processes it and its result is stored in ItemLoader.

- Next, after collecting the data, call ItemLoader.load_item() method to get the populated Item object.

- Finally, you can assign the result of the output processor to the item.

The following code demonstrates how to call input and output processors for a specific field:

```
l = ItemLoader(Product(), some_selector)
l.add_xpath("title", xpath1) # [1]
l.add_xpath("title", xpath2) #  [2]
l.add_css("title", css) # [3]
l.add_value("title", "demo") # [4]
return l.load_item() # [5]
```

Line 1: The data of title is extracted from xpath1 and passed through the input processor and its result is collected and stored in ItemLoader.

Line 2: Similarly, the title is extracted from xpath2 and passed through the same input processor and its result is added to the data collected for [1].

Line 3: The title is extracted from css selector and passed through the same input processor and the result is added to the data collected for [1] and [2].

Line 4: Next, the value "demo" is assigned and passed through the input processors.

Line 5: Finally, the data is collected internally from all the fields and passed to the output processor and the final value is assigned to the Item.

Declaring Input and Output Processors

The input and output processors are declared in the ItemLoader definition. Apart from this, they can also be specified in the **Item Field** metadata.

For example:

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

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

class Item(scrapy.Item):
    name = scrapy.Field(
        input_processor = MapCompose(remove_tags),
        output_processor = Join(),
    )
    size = scrapy.Field(
        input_processor = MapCompose(remove_tags, filter_price),
        output_processor = TakeFirst(),
    )
>>> from scrapy.loader import ItemLoader
>>> il = ItemLoader(item=Product())
>>> il.add_value('title', [u'Hello', u'<strong>world</strong>'])
>>> il.add_value('size', [u'<span>100 kg</span>'])
>>> il.load_item()
```

It displays an output as:

```
{'title': u'Hello world', 'size': u'100 kg'}
```

Item Loader Context

The Item Loader Context is a dict of arbitrary key values shared among input and output processors.

For example, assume you have a function *parse_length*:

```
def parse_length(text, loader_context):
    unit = loader_context.get('unit', 'cm')
    # You can write parsing code of length here
    return parsed_length
```

By receiving loader_context arguements, it tells the Item Loader it can receive Item Loader context. There are several ways to change the value of Item Loader context:

- Modify current active Item Loader context:

```
loader = ItemLoader (product)
loader.context ["unit"] = "mm"
```

- On Item Loader instantiation:

```
loader = ItemLoader(product, unit="mm")
```

- On Item Loader declaration for input/output processors that instantiates with Item Loader context:

```
class ProductLoader(ItemLoader):
    length_out = MapCompose(parse_length, unit="mm")
```

ItemLoader Objects

It is an object which returns a new item loader to populate the given item. It has the following class:

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

The following table shows the parameters of ItemLoader objects:

Sr. No.	Parameter & Description
1	**item** It is the item to populate by calling add_xpath(), add_css() or add_value().
2	**selector** It is used to extract data from websites.
3	**response** It is used to construct selector using default_selector_class.

Following table shows the methods of ItemLoader objects:

Sr. No.	Method & Description	Example

1	**get_value(value, *processors, **kwargs)**
	By a given processor and keyword arguments, the value is processed by get_value() method.
		`>>> from scrapy.loader.processors import TakeFirst`
		`>>> loader.get_value(u'title: demoweb', TakeFirst(), unicode.upper, re='title: (.+)')`
		`'DEMOWEB```

2	**add_value(field_name, value, *processors, **kwargs)**
	It processes the value and adds to the field where it is first passed through get_value by giving processors and keyword arguments before passing through field input processor.
		`loader.add_value('title', u'DVD')`
		`loader.add_value('colors', [u'black', u'white'])`
		`loader.add_value('length', u'80')`
		`loader.add_value('price', u'2500')`

3	**replace_value(field_name, value, *processors, **kwargs)**
	
	It replaces the collected data with a new value.
		`loader.replace_value('title', u'DVD')`
		`loader.replace_value('colors', [u'black', u'white'])`
		`loader.replace_value('length', u'80')`
		`loader.replace_value('price', u'2500')`

4	**get_xpath(xpath, *processors, **kwargs)**
	
	It is used to extract unicode strings by giving processors and keyword arguments by receiving *XPath*.
		`# HTML code: <div class="item-name">DVD</div>`
		
		`loader.get_xpath("//div[@class='item-name']")`
		`# HTML code: <div id="length">the length is 45cm</div>`
		
		`loader.get_xpath("//div[@id='length']", TakeFirst(), re="the length is (.*)")`

5	**add_xpath(field_name, xpath, *processors, **kwargs)**
	
	It receives *XPath* to the field which extracts unicode strings.
		`# HTML code: <div class="item-name">DVD</div>`
		`loader.add_xpath('name', '//div[@class="item-name"]')`
		`# HTML code: <div id="length">the length is 45cm</div>`
		`loader.add_xpath('length', '//div[@id="length"]', re='the length is (.*)')`

6	**replace_xpath(field_name, xpath, *processors, **kwargs)**
	
	It replaces the collected data using *XPath* from sites.
		`# HTML code: <div class="item-name">DVD</div>`
		`loader.replace_xpath('name', '//div[@class="item-name"]')`
		`# HTML code: <div id="length">the length is 45cm</div>`
		`loader.replace_xpath('length', '//div[@id="length"]', re='the length is (.*)')`

7	**get_css(css, *processors, **kwargs)**
	
	It receives *CSS* selector used to extract the unicode strings.
		`loader.get_css("div.item-name")`
		
		`loader.get_css("div#length", TakeFirst(), re="the length is (.*)")`

8	**add_css(field_name, css, *processors, **kwargs)**
	
	It is similar to add_value() method with one difference that it adds CSS selector to the field.
		`loader.add_css('name', 'div.item-name')`
		
		`loader.add_css('length', 'div#length', re='the length is (.*)')`

9	**replace_css(field_name, css, *processors, **kwargs)**
	
	It replaces the extracted data using CSS selector.
		`loader.replace_css('name', 'div.item-name')`
		
		`loader.replace_css('length', 'div#length', re='the length is (.*)')`

10	**load_item()**
	
	When the data is collected, this method fills the item with collected data and returns it.
		`def parse(self, response):`
		
		`l = ItemLoader(item=Product(), response=response)`
		
		`l.add_xpath('title', '//div[@class="product_title"]')`
		
		`loader.load_item()`

11	**nested_xpath(xpath)**
	
	It is used to create nested loaders with an XPath selector.
		`loader = ItemLoader(item=Item())`
		
		`loader.add_xpath('social', 'a[@class = "social"]/@href')`
		
		`loader.add_xpath('email', 'a[@class = "email"]/@href')`

12	**nested_css(css)**
	
	It is used to create nested loaders with a CSS selector.
		`loader = ItemLoader(item=Item())`
		
		`loader.add_css('social', 'a[@class = "social"]/@href')`
		
		`loader.add_css('email', 'a[@class = "email"]/@href')`

Following table shows the attributes of ItemLoader objects:

Sr. No.	Attribute & Description
1	**item** It is an object on which the Item Loader performs parsing.
2	**context** It is the current context of Item Loader that is active.
3	**default_item_class** It is used to represent the items, if not given in the constructor.
4	**default_input_processor** The fields which don't specify input processor are the only ones for which default_input_processors are used.
5	**default_output_processor** The fields which don't specify the output processor are the only ones for which default_output_processors are used.
6	**default_selector_class** It is a class used to construct the selector, if it is not given in the constructor.
7	**selector** It is an object that can be used to extract the data from sites.

Nested Loaders

It is used to create nested loaders while parsing the values from the subsection of a document. If you don't create nested loaders, you need to specify full XPath or CSS for each value that you want to extract.

For instance, assume that the data is being extracted from a header page:

```
<header>
  <a class="social" href="http://facebook.com/whatever">facebook</a>
  <a class="social" href="http://twitter.com/whatever">twitter</a>
  <a class="email" href="mailto:someone@example.com">send mail</a>
</header>
```

Next, you can create a nested loader with header selector by adding related values to the header:

```
loader = ItemLoader(item=Item())
header_loader = loader.nested_xpath('//header')
header_loader.add_xpath('social', 'a[@class = "social"]/@href')
header_loader.add_xpath('email', 'a[@class = "email"]/@href')
loader.load_item()
```

Reusing and Extending Item Loaders

Item Loaders are designed to relieve the maintenance which becomes a fundamental problem when your project acquires more spiders.

For instance, assume that a site has their product name enclosed in three dashes (e.g. ---DVD---). You can remove those dashes by reusing the default Product Item Loader, if you don’t want it in the final product names as shown in the following code:

```
from scrapy.loader.processors import MapCompose
from demoproject.ItemLoaders import DemoLoader


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


class SiteSpecificLoader(DemoLoader):
    title_in = MapCompose(strip_dashes, DemoLoader.title_in)
```

Available Built-in Processors

Following are some of the commonly used built-in processors:

class scrapy.loader.processors.Identity

It returns the original value without altering it. For example:

```
>>> from scrapy.loader.processors import Identity
>>> proc = Identity()
>>> proc(['a', 'b', 'c'])
['a', 'b', 'c']
```

class scrapy.loader.processors.TakeFirst

It returns the first value that is non-null/non-empty from the list of received values. For example:

```
>>> from scrapy.loader.processors import TakeFirst
>>> proc = TakeFirst()
>>> proc(['', 'a', 'b', 'c'])
'a'
```

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

It returns the value attached to the separator. The default separator is **u' '** and it is equivalent to the function **u' '.join**. For example:

```
>>> from scrapy.loader.processors import Join
>>> proc = Join()
>>> proc(['a', 'b', 'c'])
u'a b c'
>>> proc = Join('<br>')
>>> proc(['a', 'b', 'c'])
u'a<br>b<br>c'
```

class scrapy.loader.processors.Compose(*functions, **default_loader_context)

It is defined by a processor where each of its input value is passed to the first function, and the result of that function is passed to the second function and so on, till lthe ast function returns the final value as output.

For example:

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

class scrapy.loader.processors.MapCompose(*functions, **default_loader_context)

It is a processor where the input value is iterated and the first function is applied to each element. Next, the result of these function calls are concatenated to build new iterable that is then applied to the second function and so on, till the last function.

For example:

```
>>> def filter_scrapy(x):
        return None if x == 'scrapy' else x

>>> from scrapy.loader.processors import MapCompose
>>> proc = MapCompose(filter_scrapy, unicode.upper)
>>> proc([u'hi', u'everyone', u'im', u'pythonscrapy'])
[u'HI, u'IM', u'PYTHONSCRAPY']
```

class scrapy.loader.processors.SelectJmes(json_path)

This class queries the value using the provided json path and returns the output.

For example:

```
>>> from scrapy.loader.processors import SelectJmes, Compose, MapCompose
>>> proc = SelectJmes("hello")
>>> proc({'hello': 'scrapy'})
'scrapy'
>>> proc({'hello': {'scrapy': 'world'}})
{'scrapy': 'world'}
```

Following is the code, which queries the value by importing json:

```
>>> import json
>>> proc_single_json_str = Compose(json.loads, SelectJmes("hello"))
>>> proc_single_json_str('{"hello": "scrapy"}')
u'scrapy'
>>> proc_json_list = Compose(json.loads, MapCompose(SelectJmes('hello')))
>>> proc_json_list('[{"hello":"scrapy"}, {"world":"env"}]')
[u'scrapy']
```

8. Scrapy – Shell

Description

Scrapy shell can be used to scrap the data with error free code, without the use of spider. The main purpose of Scrapy shell is to test the extracted code, XPath, or CSS expressions. It also helps specify the web pages from which you are scraping the data.

Configuring the Shell

The shell can be configured by installing the IPython (used for interactive computing) console, which is a powerful interactive shell that gives the auto completion, colorized output, etc.

If you are working on the Unix platform, then it's better to install the IPython. You can also use bpython, if IPython is inaccessible.

You can configure the shell by setting the environment variable called SCRAPY_PYTHON_SHELL or by defining the *scrapy.cfg* file as follows:

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

Launching the Shell

Scrapy shell can be launched using the following command:

```
scrapy shell <url>
```

The *url* specifies the URL for which the data needs to be scraped.

Using the Shell

The shell provides some additional shortcuts and Scrapy objects as described in the following table:

Available Shortcuts

Shell provides the following available shortcuts in the project:

Sr. No.	Shortcut & Description
1	**shelp()** It provides the available objects and shortcuts with the help option.
2	**fetch(request_or_url)** It collects the response from the request or URL and associated objects will get updated properly.

3	**view(response)** You can view the response for the given request in the local browser for observation and to display the external link correctly, it appends a base tag to the response body.

Available Scrapy Objects

Shell provides the following available Scrapy objects in the project:

Sr. No.	Object & Description
1	**crawler** It specifies the current crawler object.
2	**spider** If there is no spider for present URL, then it will handle the URL or spider object by defining the new spider.
3	**request** It specifies the request object for the last collected page.
4	**response** It specifies the response object for the last collected page.
5	**settings** It provides the current Scrapy settings.

Example of Shell Session

Let us try scraping scrapy.org site and then begin to scrap the data from reddit.com as described.

Before moving ahead, first we will launch the shell as shown in the following command:

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

Scrapy will display the available objects while using the above URL:

```
[s] Available Scrapy objects:
[s]   crawler
[s]   item       {}
[s]   request
[s]   response   <200 http://scrapy.org>
[s]   settings
[s]   spider
[s] Useful shortcuts:
```

```
[s]   shelp()           Provides available objects and shortcuts with help
option
[s]   fetch(req_or_url) Collects the response from the request or URL and
associated objects will get update
[s]   view(response)    View the response for the given request
```

Next, begin with the working of objects, shown as follows:

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

>> fetch("http://reddit.com")
[s] Available Scrapy objects:
[s]   crawler
[s]   item       {}
[s]   request
[s]   response   <200 https://www.reddit.com/>
[s]   settings
[s]   spider
[s] Useful shortcuts:
[s]   shelp()           Shell help (print this help)
[s]   fetch(req_or_url) Fetch request (or URL) and update local objects
[s]   view(response)    View response in a browser

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

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

>> fetch(request)
[s] Available Scrapy objects:
[s]   crawler
...
```

Invoking the Shell from Spiders to Inspect Responses

You can inspect the responses which are processed from the spider, only if you are expecting to get that response.

For instance:

```
import scrapy
class SpiderDemo(scrapy.Spider):
    name = "spiderdemo"
    start_urls = [
        "http://mysite.com",
        "http://mysite1.org",
        "http://mysite2.net",
    ]

    def parse(self, response):
        # You can inspect one specific response
        if ".net" in response.url:
            from scrapy.shell import inspect_response
            inspect_response(response, self)
```

As shown in the above code, you can invoke the shell from spiders to inspect the responses using the following function:

```
scrapy.shell.inspect_response
```

Now run the spider, and you will get the following screen:

```
2016-02-08 18:15:20-0400 [scrapy] DEBUG: Crawled (200)  (referer: None)
2016-02-08 18:15:20-0400 [scrapy] DEBUG: Crawled (200)  (referer: None)
2016-02-08 18:15:20-0400 [scrapy] DEBUG: Crawled (200)  (referer: None)
[s] Available Scrapy objects:
[s]   crawler
...

>> response.url
'http://mysite2.org'
```

You can examine whether the extracted code is working using the following code:

```
>> response.xpath('//div[@class="val"]')
```

It displays the output as

```
[]
```

The above line has displayed only a blank output. Now you can invoke the shell to inspect the response as follows:

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

It displays the response as

```
True
```

9. Scrapy – Item Pipeline

Description

Item Pipeline is a method where the scrapped items are processed. When an item is sent to the Item Pipeline, it is scraped by a spider and processed using several components, which are executed sequentially.

Whenever an item is received, it decides either of the following action:

- Keep processing the item.
- Drop it from pipeline.
- Stop processing the item.

Item pipelines are generally used for the following purposes:

- Storing scraped items in database.
- If the received item is repeated, then it will drop the repeated item.
- It will check whether the item is with targeted fields.
- Clearing HTML data.

Syntax

You can write the Item Pipeline using the following method:

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

The above method contains following parameters:

- Item (item object or dictionary) - It specifies the scraped item.
- spider (spider object) - The spider which scraped the item.

You can use additional methods given in the following table:

Sr. No.	Method & Description	Parameters
1	**open_spider(*self, spider*)** It is selected when spider is opened.	spider (spider object) - It refers to the spider which was opened.
2	**close_spider(*self, spider*)** It is selected when spider is closed.	spider (spider object) - It refers to the spider which was closed.
3	**from_crawler(*cls, crawler*)** With the help of crawler, the pipeline can access the core components such as signals and settings of Scrapy.	crawler (Crawler object) – It refers to the crawler that uses this pipeline.

Example

Following are the examples of item pipeline used in different concepts.

Dropping Items with No Tag

In the following code, the pipeline balances the *(price)* attribute for those items that do not include VAT *(excludes_vat attribute)* and ignore those items which do not have a price tag:

```
from Scrapy.exceptions import DropItem


class PricePipeline(object):


    vat = 2.25


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

Writing Items to a JSON File

The following code will store all the scraped items from all spiders into a single **items.jl** file, which contains one item per line in a serialized form in JSON format. The **JsonWriterPipeline** class is used in the code to show how to write item pipeline:

```
import json


class JsonWriterPipeline(object):


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


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

Writing Items to MongoDB

You can specify the MongoDB address and database name in Scrapy settings and MongoDB collection can be named after the item class. The following code describes how to use **from_crawler()** method to collect the resources properly:

```
import pymongo


class MongoPipeline(object):


    collection_name = 'Scrapy_list'


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


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


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


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


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

Duplicating Filters

A filter will check for the repeated items and it will drop the already processed items. In the following code, we have used a unique id for our items, but spider returns many items with the same id:

```
from scrapy.exceptions import DropItem

class DuplicatesPipeline(object):

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

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

Activating an Item Pipeline

You can activate an Item Pipeline component by adding its class to the *ITEM_PIPELINES* setting as shown in the following code. You can assign integer values to the classes in the order in which they run (the order can be lower valued to higher valued classes) and values will be in the 0-1000 range.

```
ITEM_PIPELINES = {
    'myproject.pipelines.PricePipeline': 100,
    'myproject.pipelines.JsonWriterPipeline': 600,
}
```

10. Scrapy – Feed Exports

Description

Feed exports is a method of storing the data scraped from the sites, that is generating a **"export file"**.

Serialization Formats

Using multiple serialization formats and storage backends, Feed Exports use Item exporters and generates a feed with scraped items.

The following table shows the supported formats:

Sr. No.	Format & Description
1	**JSON** FEED_FORMAT is *json* Exporter used is *class scrapy.exporters.JsonItemExporter*
2	**JSON lines** FEED_FROMAT is *jsonlines* Exporter used is *class scrapy.exporters.JsonLinesItemExporter*
3	**CSV** FEED_FORMAT is *CSV* Exporter used is *class scrapy.exporters.CsvItemExporter*
4	**XML** FEED_FORMAT is *xml* Exporter used is *class scrapy.exporters.XmlItemExporter*

Using **FEED_EXPORTERS** settings, the supported formats can also be extended:

Sr. No.	Format & Description
1	**Pickle** FEED_FORMAT is pickle Exporter used is *class scrapy.exporters.PickleItemExporter*
2	**Marshal** FEED_FORMAT is marshal Exporter used is *class scrapy.exporters.MarshalItemExporter*

Storage Backends

Storage backend defines where to store the feed using URI.

Following table shows the supported storage backends:

Sr. No.	Storage Backend & Description
1	**Local filesystem** URI scheme is *file* and it is used to store the feeds.
2	**FTP** URI scheme is *ftp* and it is used to store the feeds.
3	**S3** URI scheme is *S3* and the feeds are stored on Amazon S3. External libraries ***botocore*** or ***boto*** are required.
4	**Standard output** URI scheme is *stdout* and the feeds are stored to the standard output.

Storage URI Parameters

Following are the parameters of storage URL, which gets replaced while the feed is being created:

- %(time)s: This parameter gets replaced by a timestamp.
- %(name)s: This parameter gets replaced by spider name.

Settings

Following table shows the settings using which Feed exports can be configured:

Sr. No.	Setting & Description
1	**FEED_URI** It is the URI of the export feed used to enable feed exports.
2	**FEED_FORMAT** It is a serialization format used for the feed.
3	**FEED_EXPORT_FIELDS** It is used for defining fields which needs to be exported.
4	**FEED_STORE_EMPTY** It defines whether to export feeds with no items.

5	**FEED_STORAGES** It is a dictionary with additional feed storage backends.
6	**FEED_STORAGES_BASE** It is a dictionary with built-in feed storage backends.
7	**FEED_EXPORTERS** It is a dictionary with additional feed exporters.
8	**FEED_EXPORTERS_BASE** It is a dictionary with built-in feed exporters.

11. Scrapy – Requests & Responses

Description

Scrapy can crawl websites using the **Request** and **Response** objects. The request objects pass over the system, uses the spiders to execute the request and get back to the request when it returns a response object.

Request Objects

The request object is a HTTP request that generates a response. It has the following class:

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

Following table shows the parameters of Request objects:

Sr. No.	Parameter & Description
1	**url** It is a string that specifies the URL request.
2	**callback** It is a callable function which uses the response of the request as first parameter.
3	**method** It is a string that specifies the HTTP method request.
4	**headers** It is a dictionary with request headers.
5	**body** It is a string or unicode that has a request body.
6	**cookies** It is a list containing request cookies.
7	**meta** It is a dictionary that contains values for metadata of the request.
8	**encoding** It is a string containing utf-8 encoding used to encode URL.
9	**priority** It is an integer where the scheduler uses priority to define the order to process requests.

10	**dont_filter** It is a boolean specifying that the scheduler should not filter the request.
11	**errback** It is a callable function to be called when an exception while processing a request is raised.

Passing Additional Data to Callback Functions

The callback function of a request is called when the response is downloaded as its first parameter.

For example:

```
def parse_page1(self, response):
    return scrapy.Request("http://www.something.com/some_page.html",
                          callback=self.parse_page2)
    def parse_page2(self, response):
        self.logger.info("%s page visited", response.url)
```

You can use **Request.meta** attribute, if you want to pass arguments to callable functions and receive those arguments in the second callback as shown in the following example:

```
def parse_page1(self, response):
    item = DemoItem()
    item['foremost_link'] = response.url
    request = scrapy.Request("http://www.something.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_link'] = response.url
        return item
```

Using errbacks to Catch Exceptions in Request Processing

The errback is a callable function to be called when an exception while processing a request is raised.

The following example demonstrates this.

```
import scrapy
```

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

class DemoSpider(scrapy.Spider):
    name = "demo"
    start_urls = [
        "http://www.httpbin.org/",              # HTTP 200 expected
        "http://www.httpbin.org/status/404",    # Webpage not found
        "http://www.httpbin.org/status/500",    # Internal server error
        "http://www.httpbin.org:12345/",        # timeout expected
        "http://www.httphttpbinbin.org/",       # DNS error expected
    ]

    def start_requests(self):
        for u in self.start_urls:
            yield scrapy.Request(u, callback=self.parse_httpbin,
                                 errback=self.errback_httpbin,
                                 dont_filter=True)

    def parse_httpbin(self, response):
        self.logger.info('Recieved response from {}'.format(response.url))
        # ...

    def errback_httpbin(self, failure):
        # logs failures
        self.logger.error(repr(failure))

        if failure.check(HttpError):
           response = failure.value.response
           self.logger.error("HttpError occurred on %s", response.url)

        elif failure.check(DNSLookupError):
             request = failure.request
             self.logger.error("DNSLookupError occurred on %s", request.url)
             elif failure.check(TimeoutError, TCPTimedOutError):
             request = failure.request
             self.logger.error("TimeoutError occurred on %s", request.url)
```

Request.meta Special Keys

The request.meta special keys is a list of special meta keys identified by Scrapy.

Following table shows some of the keys of Request.meta:

Sr. No.	Key & Description
1	**dont_redirect** It is a key when set to true, does not redirect the request based on the status of the response.
2	**dont_retry** It is a key when set to true, does not retry the failed requests and will be ignored by the middleware.
3	**handle_httpstatus_list** It is a key that defines which response codes per-request basis can be allowed.
4	**handle_httpstatus_all** It is a key used to allow any response code for a request by setting it to *true*.
5	**dont_merge_cookies** It is a key used to avoid merging with the existing cookies by setting it to *true*.
6	**cookiejar** It is a key used to keep multiple cookie sessions per spider.
7	**dont_cache** It is a key used to avoid caching HTTP requests and response on each policy.
8	**redirect_urls** It is a key which contains URLs through which the requests pass.
9	**bindaddress** It is the IP of the outgoing IP address that can be used to perform the request.
10	**dont_obey_robotstxt** It is a key when set to true, does not filter the requests prohibited by the robots.txt exclusion standard, even if ROBOTSTXT_OBEY is enabled.

11	**download_timeout** It is used to set timeout (in secs) per spider for which the downloader will wait before it times out.
12	**download_maxsize** It is used to set maximum size (in bytes) per spider, which the downloader will download.
13	**proxy** Proxy can be set for *Request* objects to set HTTP proxy for the use of requests.

Request Subclasses

You can implement your own custom functionality by subclassing the request class. The built-in request subclasses are as follows:

FormRequest Objects

The FormRequest class deals with HTML forms by extending the base request. It has the following class:

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

Following is the parameter:

formdata: It is a dictionary having HTML form data that is assigned to the body of the request.

Note: Remaining parameters are the same as *request* class and is explained in **Request Objects** section.

The following class methods are supported by **FormRequest** objects in addition to request methods:

```
classmethod from_response(response[, formname=None, formnumber=0,
formdata=None, formxpath=None, formcss=None, clickdata=None,
dont_click=False, ...])
```

The following table shows the parameters of the above class:

Sr. No.	Parameter & Description
1	**response** It is an object used to pre-populate the form fields using HTML form of response.
2	**formname** It is a string where the form having name attribute will be used, if specified.

3	**formnumber** It is an integer of forms to be used when there are multiple forms in the response.
4	**formdata** It is a dictionary of fields in the form data used to override.
5	**formxpath** It is a string when specified, the form matching the xpath is used.
6	**formcss** It is a string when specified, the form matching the css selector is used.
7	**clickdata** It is a dictionary of attributes used to observe the clicked control.
8	**dont_click** The data from the form will be submitted without clicking any element, when set to true.

Examples

Following are some of the request usage examples:

Using FormRequest to send data via HTTP POST

The following code demonstrates how to return **FormRequest** object when you want to duplicate HTML form POST in your spider:

```
return [FormRequest(url="http://www.something.com/post/action",
                         formdata={'firstname': 'John', 'lastname': 'dave'},
                         callback=self.after_post)]
```

Using FormRequest.from_response() to simulate a user login

Normally, websites use elements through which it provides pre-populated form fields. The **FormRequest.form_response()** method can be used when you want these fields to be automatically populate while scraping.

The following example demonstrates this.

```
import scrapy


class DemoSpider(scrapy.Spider):
    name = 'demo'
    start_urls = ['http://www.something.com/users/login.php']


    def parse(self, response):
        return scrapy.FormRequest.from_response(
            response,
            formdata={'username': 'admin', 'password': 'confidential'},
            callback=self.after_login
        )

    def after_login(self, response):
        if "authentication failed" in response.body:
        self.logger.error("Login failed")
            return

      # You can continue scraping here
```

Response Objects

It is an object indicating HTTP response that is fed to the spiders to process. It has the following class:

```
class scrapy.http.Response(url[, status=200, headers, body, flags])
```

The following table shows the parameters of Response objects:

Sr. No.	Parameter & Description
1	**url** It is a string that specifies the URL response.
2	**status** It is an integer that contains HTTP status response.
3	**headers** It is a dictionary containing response headers.
4	**body** It is a string with response body.
5	**flags** It is a list containing flags of response.

Response Subclasses

You can implement your own custom functionality by subclassing the response class. The built-in response subclasses are as follows:

TextResponse objects

TextResponse objects are used for binary data such as images, sounds, etc. which has the ability to encode the base Response class. It has the following class:

```
class scrapy.http.TextResponse(url[, encoding[,status=200, headers, body,
flags]])
```

Following is the parameter:

encoding: It is a string with encoding that is used to encode a response.

Note: Remaining parameters are same as *response* class and is explained in **Response Objects** section.

The following table shows the attributes supported by *TextResponse* object in addition to response methods:

Sr. No.	Attribute & Description
1	**text** It is a response body, where response.text can be accessed multiple times.
2	**encoding** It is a string containing encoding for response.
3	**selector** It is an attribute instantiated on first access and uses response as target.

The following table shows the methods supported by *TextResponse* objects in addition to *response* methods:

Sr. No.	Method & Description
1	**xpath (query)** It is a shortcut to TextResponse.selector.xpath(query).
2	**css (query)** It is a shortcut to TextResponse.selector.css(query).
3	**body_as_unicode()** It is a response body available as a method, where response.text can be accessed multiple times.

HtmlResponse Objects

It is an object that supports encoding and auto-discovering by looking at the *meta http-equiv* attribute of HTML. Its parameters are the same as *response* class and is explained in *Response objects* section. It has the following class:

```
class scrapy.http.HtmlResponse(url[,status=200, headers, body, flags])
```

XmlResponse Objects

It is an object that supports encoding and auto-discovering by looking at the XML line. Its parameters are the same as *response* class and is explained in *Response objects* section. It has the following class:

```
class scrapy.http.XmlResponse(url[, status=200, headers, body, flags])
```

12. Scrapy – Link Extractors

Description

As the name itself indicates, Link Extractors are the objects that are used to extract links from web pages using **scrapy.http.Response** objects. In Scrapy, there are built-in extractors such as **scrapy.linkextractors** import **LinkExtractor.** You can customize your own link extractor according to your needs by implementing a simple interface.

Every link extractor has a public method called **extract_links** which includes a *Response* object and returns a list of scrapy.link.Link objects. You can instantiate the link extractors only once and call the extract_links method various times to extract links with different responses. The CrawlSpiderclass uses link extractors with a set of rules whose main purpose is to extract links.

Built-in Link Extractor's Reference

Normally link extractors are grouped with Scrapy and are provided in scrapy.linkextractors module. By default, the link extractor will be LinkExtractor which is equal in functionality with LxmlLinkExtractor*:*

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

LxmlLinkExtractor

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

The *LxmlLinkExtractor* is a highly recommended link extractor, because it has handy filtering options and it is used with lxml's robust HTMLParser.

Sr. No.	Parameters	Description
1	**allow** (a regular expression (or list of))	It allows a single expression or group of expressions that should match the URL which is to be extracted. If it is not mentioned, it will match all the links.
2	**deny** (a regular expression (or list of))	It blocks or excludes a single expression or group of expressions that should match the URL which is not to be extracted. If it is not mentioned or left empty, then it will not eliminate the undesired links.
3	**allow_domains** (str or list)	It allows a single string or list of strings that should match the domains from which the links are to be extracted.

4	**deny_domains** *(str or list)*	It blocks or excludes a single string or list of strings that should match the domains from which the links are not to be extracted.
5	**deny_extensions** *(list)*	It blocks the list of strings with the extensions when extracting the links. If it is not set, then by default it will be set to *IGNORED_EXTENSIONS* which contains pre-defined list in *scrapy.linkextractors* package.
6	**restrict_xpaths** *(str or list)*	It is an XPath list region from where the links are to be extracted from the response. If given, the links will be extracted only from the text, which is selected by XPath.
7	**restrict_css** *(str or list)*	It behaves similar to *restrict_xpaths* parameter which will extract the links from the CSS selected regions inside the response.
8	**tags** *(str or list)*	A single tag or a list of tags that should be considered when extracting the links. By default, it will be *('a', 'area')*.
9	**attrs** *(list)*	A single attribute or list of attributes should be considered while extracting links. By default, it will be *('href',)*.
10	**canonicalize** *(boolean)*	The extracted url is brought to standard form using *scrapy.utils.url.canonicalize_url*. By default, it will be *True*.
11	**unique** *(boolean)*	It will be used if the extracted links are repeated.
12	**process_value** *(callable)*	It is a function which receives a value from scanned tags and attributes. The value received may be altered and returned or else nothing will be returned to reject the link. If not used, by default it will be *lambda x: x*.

Example

The following code is used to extract the links:

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

The following code function can be used in process_value:

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

13. Scrapy – Settings

Description

The behavior of Scrapy components can be modified using Scrapy settings. The settings can also select the Scrapy project that is currently active, in case you have multiple Scrapy projects.

Designating the Settings

You must notify Scrapy which setting you are using when you scrap a website. For this, environment variable **SCRAPY_SETTINGS_MODULE** should be used and its value should be in Python path syntax.

Populating the Settings

The following table shows some of the mechanisms by which you can populate the settings:

Sr. No.	Mechanism & Description
1	**Command line options** Here, the arguments that are passed takes highest precedence by overriding other options. The **-s** is used to override one or more settings. `scrapy crawl myspider -s LOG_FILE=scrapy.log`

2	**Settings per-spider**
	Spiders can have their own settings that overrides the project ones by using attribute custom_settings.
	`class DemoSpider(scrapy.Spider):`
	`    name = 'demo'`
	
	`    custom_settings = {`
	`        'SOME_SETTING': 'some value',`
	`    }`

3	**Project settings module** Here, you can populate your custom settings such as adding or modifying the settings in the settings.py file.
4	**Default settings per-command** Each Scrapy tool command defines its own settings in the default_settings attribute, to override the global default settings.
5	**Default global settings** These settings are found in the scrapy.settings.default_settings module.

Access Settings

They are available through *self.settings* and set in the base spider after it is initialized.

The following example demonstrates this.

```
class DemoSpider(scrapy.Spider):
    name = 'demo'
    start_urls = ['http://example.com']

    def parse(self, response):
        print("Existing settings: %s" % self.settings.attributes.keys())
```

To use settings before initializing the spider, you must override *from_crawler* method in the *_init_()* method of your spider. You can access settings through attribute *scrapy.crawler.Crawler.settings* passed to *from_crawler* method.

The following example demonstrates this.

```
class MyExtension(object):
    def __init__(self, log_is_enabled=False):
        if log_is_enabled:
            print("Enabled log")

    @classmethod
    def from_crawler(cls, crawler):
        settings = crawler.settings
        return cls(settings.getbool('LOG_ENABLED'))
```

Rationale for Setting Names

Setting names are added as a prefix to the component they configure. For example, for robots.txt extension, the setting names can be ROBOTSTXT_ENABLED, ROBOTSTXT_OBEY, ROBOTSTXT_CACHEDIR, etc.

Built-in Settings Reference

The following table shows the built-in settings of Scrapy:

Sr. No.	Setting & Description
1	**AWS_ACCESS_KEY_ID** It is used to access Amazon Web Services. Default value: None
2	**AWS_SECRET_ACCESS_KEY** It is used to access Amazon Web Services. Default value: None
3	**BOT_NAME** It is the name of bot that can be used for constructing User-Agent. Default value: 'scrapybot'
4	**CONCURRENT_ITEMS** Maximum number of existing items in the Item Processor used to process parallely. Default value: 100
5	**CONCURRENT_REQUESTS** Maximum number of existing requests which Scrapy downloader performs. Default value: 16
6	**CONCURRENT_REQUESTS_PER_DOMAIN** Maximum number of existing requests that perform simultaneously for any single domain. Default value: 8
7	**CONCURRENT_REQUESTS_PER_IP** Maximum number of existing requests that performs simultaneously to any single IP. Default value: 0

8	**DEFAULT_ITEM_CLASS** It is a class used to represent items. Default value: 'scrapy.item.Item'

9	**DEFAULT_REQUEST_HEADERS**
	
	It is a default header used for HTTP requests of Scrapy.
	Default value:
	`{`
	`   'Accept':`
	`'text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8',`
	`'Accept-Language': 'en',`
	`}`

10	**DEPTH_LIMIT** The maximum depth for a spider to crawl any site. Default value: 0
11	**DEPTH_PRIORITY** It is an integer used to alter the priority of request according to the depth. Default value: 0
12	**DEPTH_STATS** It states whether to collect depth stats or not. Default value: True
13	**DEPTH_STATS_VERBOSE** This setting when enabled, the number of requests is collected in stats for each verbose depth. Default value: False
14	**DNSCACHE_ENABLED** It is used to enable DNS in memory cache. Default value: True

15	**DNSCACHE_SIZE** It defines the size of DNS in memory cache. Default value: 10000
16	**DNS_TIMEOUT** It is used to set timeout for DNS to process the queries. Default value: 60
17	**DOWNLOADER** It is a downloader used for the crawling process. Default value: 'scrapy.core.downloader.Downloader'
18	**DOWNLOADER_MIDDLEWARES** It is a dictionary holding downloader middleware and their orders. Default value: {}

19	**DOWNLOADER_MIDDLEWARES_BASE**
	
	It is a dictionary holding downloader middleware that is enabled by default.
	
	Default value: {
	'scrapy.downloadermiddlewares.robotstxt.RobotsTxtMiddleware': 100, }

20	**DOWNLOADER_STATS** This setting is used to enable the downloader stats. Default value: True
21	**DOWNLOAD_DELAY** It defines the total time for downloader before it downloads the pages from the site. Default value: 0
22	**DOWNLOAD_HANDLERS** It is a dictionary with download handlers. Default value: {}
23	**DOWNLOAD_HANDLERS_BASE** It is a dictionary with download handlers that is enabled by default. Default value: { 'file': 'scrapy.core.downloader.handlers.file.FileDownloadHandler', }

24	**DOWNLOAD_TIMEOUT** It is the total time for downloader to wait before it times out. Default value: 180
25	**DOWNLOAD_MAXSIZE** It is the maximum size of response for the downloader to download. Default value: 1073741824 (1024MB)
26	**DOWNLOAD_WARNSIZE** It defines the size of response for downloader to warn. Default value: 33554432 (32MB)
27	**DUPEFILTER_CLASS** It is a class used for detecting and filtering of requests that are duplicate. Default value: 'scrapy.dupefilters.RFPDupeFilter'
28	**DUPEFILTER_DEBUG** This setting logs all duplicate filters when set to true. Default value: False
29	**EDITOR** It is used to edit spiders using the edit command. Default value: Depends on the environment
30	**EXTENSIONS** It is a dictionary having extensions that are enabled in the project. Default value: {}
31	**EXTENSIONS_BASE** It is a dictionary having built-in extensions. Default value: { 'scrapy.extensions.corestats.CoreStats': 0, }
32	**FEED_TEMPDIR** It is a directory used to set the custom folder where crawler temporary files can be stored.

33	**ITEM_PIPELINES** It is a dictionary having pipelines. Default value: {}
34	**LOG_ENABLED** It defines if the logging is to be enabled. Default value: True
35	**LOG_ENCODING** It defines the type of encoding to be used for logging. Default value: 'utf-8'
36	**LOG_FILE** It is the name of the file to be used for the output of logging. Default value: None
37	**LOG_FORMAT** It is a string using which the log messages can be formatted. Default value: '%(asctime)s [%(name)s] %(levelname)s: %(message)s'
38	**LOG_DATEFORMAT** It is a string using which date/time can be formatted. Default value: '%Y-%m-%d %H:%M:%S'
39	**LOG_LEVEL** It defines minimum log level. Default value: 'DEBUG'
40	**LOG_STDOUT** This setting if set to *true*, all your process output will appear in the log. Default value: False
41	**MEMDEBUG_ENABLED** It defines if the memory debugging is to be enabled. Default Value: False

42	**MEMDEBUG_NOTIFY** It defines the memory report that is sent to a particular address when memory debugging is enabled. Default value: []
43	**MEMUSAGE_ENABLED** It defines if the memory usage is to be enabled when a Scrapy process exceeds a memory limit. Default value: False
44	**MEMUSAGE_LIMIT_MB** It defines the maximum limit for the memory (in megabytes) to be allowed. Default value: 0
45	**MEMUSAGE_CHECK_INTERVAL_SECONDS** It is used to check the present memory usage by setting the length of the intervals. Default value: 60.0
46	**MEMUSAGE_NOTIFY_MAIL** It is used to notify with a list of emails when the memory reaches the limit. Default value: False
47	**MEMUSAGE_REPORT** It defines if the memory usage report is to be sent on closing each spider. Default value: False
48	**MEMUSAGE_WARNING_MB** It defines a total memory to be allowed before a warning is sent. Default value: 0
49	**NEWSPIDER_MODULE** It is a module where a new spider is created using *genspider* command. Default value: ''

50	**RANDOMIZE_DOWNLOAD_DELAY** It defines a random amount of time for a Scrapy to wait while downloading the requests from the site. Default value: True
51	**REACTOR_THREADPOOL_MAXSIZE** It defines a maximum size for the reactor threadpool. Default value: 10
52	**REDIRECT_MAX_TIMES** It defines how many times a request can be redirected. Default value: 20
53	**REDIRECT_PRIORITY_ADJUST** This setting when set, adjusts the redirect priority of a request. Default value: +2
54	**RETRY_PRIORITY_ADJUST** This setting when set, adjusts the retry priority of a request. Default value: -1
55	**ROBOTSTXT_OBEY** Scrapy obeys robots.txt policies when set to *true*. Default value: False
56	**SCHEDULER** It defines the scheduler to be used for crawl purpose. Default value: 'scrapy.core.scheduler.Scheduler'
57	**SPIDER_CONTRACTS** It is a dictionary in the project having spider contracts to test the spiders. Default value: {}

58	**SPIDER_CONTRACTS_BASE**
	It is a dictionary holding Scrapy contracts which is enabled in Scrapy by default.
	
	Default value:
	`{`
	`  'scrapy.contracts.default.UrlContract' : 1,`
	`  'scrapy.contracts.default.ReturnsContract': 2,`
	`}`

59	**SPIDER_LOADER_CLASS** It defines a class which implements *SpiderLoader API* to load spiders. Default value: 'scrapy.spiderloader.SpiderLoader'
60	**SPIDER_MIDDLEWARES** It is a dictionary holding spider middlewares. Default value: {}

61	**SPIDER_MIDDLEWARES_BASE**
	
	It is a dictionary holding spider middlewares that is enabled in Scrapy by default.
	
	Default value:
	`{`
	`  'scrapy.spidermiddlewares.httperror.HttpErrorMiddleware': 50,`
	`}`

62	**SPIDER_MODULES** It is a list of modules containing spiders which Scrapy will look for. Default value: []
63	**STATS_CLASS** It is a class which implements *Stats Collector API* to collect stats. Default value: 'scrapy.statscollectors.MemoryStatsCollector'
64	**STATS_DUMP** This setting when set to *true*, dumps the stats to the log. Default value: True
65	**STATSMAILER_RCPTS** Once the spiders finish scraping, Scrapy uses this setting to send the stats. Default value: []
66	**TELNETCONSOLE_ENABLED** It defines whether to enable the telnetconsole. Default value: True

67	**TELNETCONSOLE_PORT** It defines a port for telnet console. Default value: [6023, 6073]
68	**TEMPLATES_DIR** It is a directory containing templates that can be used while creating new projects. Default value: *templates* directory inside scrapy module
69	**URLLENGTH_LIMIT** It defines the maximum limit of the length for URL to be allowed for crawled URLs. Default value: 2083
70	**USER_AGENT** It defines the user agent to be used while crawling a site. Default value: "Scrapy/VERSION (+http://scrapy.org)"

For other Scrapy settings, go to this link.

Other Settings

The following table shows other settings of Scrapy:

Sr. No.	Setting & Description
1	**AJAXCRAWL_ENABLED** It is used for enabling the large crawls. Default value: False
2	**AUTOTHROTTLE_DEBUG** It is enabled to see how throttling parameters are adjusted in real time, which displays stats on every received response. Default value: False

3	**AUTOTHROTTLE_ENABLED** It is used to enable AutoThrottle extension. Default value: False
4	**AUTOTHROTTLE_MAX_DELAY** It is used to set the maximum delay for download in case of high latencies. Default value: 60.0
5	**AUTOTHROTTLE_START_DELAY** It is used to set the initial delay for download. Default value: 5.0
6	**AUTOTHROTTLE_TARGET_CONCURRENCY** It defines the average number of requests for a Scrapy to send parallelly to remote sites. Default value: 1.0
7	**CLOSESPIDER_ERRORCOUNT** It defines total number of errors that should be recieved before the spider is closed. Default value: 0
8	**CLOSESPIDER_ITEMCOUNT** It defines a total number of items before closing the spider. Default value: 0
9	**CLOSESPIDER_PAGECOUNT** It defines the maximum number of responses to crawl before spider closes. Default value: 0
10	**CLOSESPIDER_TIMEOUT** It defines the amount of time (in sec) for a spider to close. Default value: 0
11	**COMMANDS_MODULE** It is used when you want to add custom commands in your project. Default value: ''

12	**COMPRESSION_ENABLED** It indicates that the compression middleware is enabled. Default value: True
13	**COOKIES_DEBUG** If set to *true*, all the cookies sent in requests and received in responses are logged. Default value: False
14	**COOKIES_ENABLED** It indicates that cookies middleware is enabled and sent to web servers. Default value: True
15	**FILES_EXPIRES** It defines the delay for the file expiration. Default value: 90 days
16	**FILES_RESULT_FIELD** It is set when you want to use other field names for your processed files.
17	**FILES_STORE** It is used to store the downloaded files by setting it to a valid value.
18	**FILES_STORE_S3_ACL** It is used to modify the ACL policy for the files stored in Amazon S3 bucket. Default value: private
19	**FILES_URLS_FIELD** It is set when you want to use other field name for your files URLs.
20	**HTTPCACHE_ALWAYS_STORE** Spider will cache the pages thoroughly if this setting is enabled. Default value: False
21	**HTTPCACHE_DBM_MODULE** It is a database module used in DBM storage backend. Default value: 'anydbm'

22	**HTTPCACHE_DIR** It is a directory used to enable and store the HTTP cache. Default value: 'httpcache'
23	**HTTPCACHE_ENABLED** It indicates that HTTP cache is enabled. Default value: False
24	**HTTPCACHE_EXPIRATION_SECS** It is used to set the expiration time for HTTP cache. Default value: 0
25	**HTTPCACHE_GZIP** This setting if set to *true*, all the cached data will be compressed with gzip. Default value: False
26	**HTTPCACHE_IGNORE_HTTP_CODES** It states that HTTP responses should not be cached with HTTP codes. Default value: []
27	**HTTPCACHE_IGNORE_MISSING** This setting if enabled, the requests will be ignored if not found in the cache. Default value: False
28	**HTTPCACHE_IGNORE_RESPONSE_CACHE_CONTROLS** It is a list containing cache controls to be ignored. Default value: []
29	**HTTPCACHE_IGNORE_SCHEMES** It states that HTTP responses should not be cached with URI schemes. Default value: ['file']
30	**HTTPCACHE_POLICY** It defines a class implementing cache policy. Default value: 'scrapy.extensions.httpcache.DummyPolicy'
31	**HTTPCACHE_STORAGE** It is a class implementing the cache storage. Default value: 'scrapy.extensions.httpcache.FilesystemCacheStorage'

32	**HTTPERROR_ALLOWED_CODES** It is a list where all the responses are passed with non-200 status codes. Default value: []
33	**HTTPERROR_ALLOW_ALL** This setting when enabled, all the responses are passed despite of its status codes. Default value: False
34	**HTTPPROXY_AUTH_ENCODING** It is used to authenticate the proxy on *HttpProxyMiddleware*. Default value: "latin-1"
35	**IMAGES_EXPIRES** It defines the delay for the images expiration. Default value: 90 days
36	**IMAGES_MIN_HEIGHT** It is used to drop images that are too small using minimum size.
37	**IMAGES_MIN_WIDTH** It is used to drop images that are too small using minimum size.
38	**IMAGES_RESULT_FIELD** It is set when you want to use other field name for your processed images.
39	**IMAGES_STORE** It is used to store the downloaded images by setting it to a valid value.
40	**IMAGES_STORE_S3_ACL** It is used to modify the ACL policy for the images stored in Amazon S3 bucket. Default value: private
41	**IMAGES_THUMBS** It is set to create the thumbnails of downloaded images.
42	**IMAGES_URLS_FIELD** It is set when you want to use other field name for your images URLs.

43	**MAIL_FROM** The sender uses this setting to send the emails. Default value: 'scrapy@localhost'
44	**MAIL_HOST** It is a SMTP host used to send emails. Default value: 'localhost'
45	**MAIL_PASS** It is a password used to authenticate SMTP. Default value: None
46	**MAIL_PORT** It is a SMTP port used to send emails. Default value: 25
47	**MAIL_SSL** It is used to implement connection using SSL encrypted connection. Default value: False
48	**MAIL_TLS** When enabled, it forces connection using STARTTLS. Default value: False
49	**MAIL_USER** It defines a user to authenticate SMTP. Default value: None
50	**METAREFRESH_ENABLED** It indicates that meta refresh middleware is enabled. Default value: True
51	**METAREFRESH_MAXDELAY** It is a maximum delay for a meta-refresh to redirect. Default value: 100
52	**REDIRECT_ENABLED** It indicates that the redirect middleware is enabled. Default value: True

53	**REDIRECT_MAX_TIMES** It defines the maximum number of times for a request to redirect. Default value: 20
54	**REFERER_ENABLED** It indicates that referrer middleware is enabled. Default value: True
55	**RETRY_ENABLED** It indicates that the retry middleware is enabled. Default value: True
56	**RETRY_HTTP_CODES** It defines which HTTP codes are to be retried. Default value: [500, 502, 503, 504, 408]
57	**RETRY_TIMES** It defines maximum number of times for retry. Default value: 2
58	**TELNETCONSOLE_HOST** It defines an interface on which the telnet console must listen. Default value: '127.0.0.1'
59	**TELNETCONSOLE_PORT** It defines a port to be used for telnet console. Default value: [6023, 6073]

14. Scrapy – Exceptions

Description

The irregular events are referred to as exceptions. In Scrapy, exceptions are raised due to reasons such as missing configuration, dropping item from the item pipeline, etc. Following is the list of exceptions mentioned in Scrapy and their application.

DropItem

Item Pipeline utilizes this exception to stop processing of the item at any stage. It can be written as:

```
exception (scrapy.exceptions.DropItem)
```

CloseSpider

This exception is used to stop the spider using the callback request. It can be written as:

```
exception (scrapy.exceptions.CloseSpider)(reason='cancelled')
```

It contains parameter called *reason (str)* which specifies the reason for closing.

For instance, the following code shows this exception usage:

```
def parse_page(self, response):
    if 'Bandwidth exceeded' in response.body:
        raise CloseSpider('bandwidth_exceeded')
```

IgnoreRequest

This exception is used by scheduler or downloader middleware to ignore a request. It can be written as:

```
exception (scrapy.exceptions.IgnoreRequest)
```

NotConfigured

It indicates a missing configuration situation and should be raised in a component constructor.

```
exception (scrapy.exceptions.NotConfigured)
```

This exception can be raised, if any of the following components are disabled.

- Extensions
- Item pipelines
- Downloader middlewares
- Spider middlewares

NotSupported

This exception is raised when any feature or method is not supported. It can be written as:

```
exception (scrapy.exceptions.NotSupported)
```

Scrapy Live Project

15. Scrapy – Create a Project

Description

To scrap the data from web pages, first you need to create the Scrapy project where you will be storing the code. To create a new directory, run the following command:

```
scrapy startproject first_scrapy
```

The above code will create a directory with name *first_scrapy* and it will contain the following structure:

```
first_scrapy/
scrapy.cfg            # deploy configuration file
first_scrapy/         # project's Python module, you'll import your code from here
__init__.py
items.py              # project items file
pipelines.py          # project pipelines file
settings.py           # project settings file
spiders/              # a directory where you'll later put your spiders
__init__.py
```

16. Scrapy – Define an Item

Description

Items are the containers used to collect the data that is scrapped from the websites. You must start your spider by defining your *Item*. To define items, edit **items.py** file found under directory **first_scrapy** (custom directory). The *items.py* looks like the following:

```
import scrapy


class First_scrapyItem(scrapy.Item):
    # define the fields for your item here like:
    # name = scrapy.Field()
```

The *MyItem* class inherits from *Item* containing a number of pre-defined objects that Scrapy has already built for us. For instance, if you want to extract the name, URL, and description from the sites, you need to define the fields for each of these three attributes.

Hence, let's add those items that we want to collect:

```
from scrapy.item import Item, Field


class First_scrapyItem(scrapy.Item):
    name = scrapy.Field()
    url = scrapy.Field()
    desc = scrapy.Field()
```

17. Scrapy – First Spider

Description

Spider is a class that defines initial URL to extract the data from, how to follow pagination links and how to extract and parse the fields defined in the **items.py**. Scrapy provides different types of spiders each of which gives a specific purpose.

Create a file called "**first_spider.py**" under the first_scrapy/spiders directory, where we can tell Scrapy how to find the exact data we're looking for. For this, you must define some attributes:

- **name**: It defines the unique name for the spider.
- **allowed_domains**: It contains the base URLs for the spider to crawl.
- **start-urls**: A list of URLs from where the spider starts crawling.
- **parse()**: It is a method that extracts and parses the scraped data.

The following code demonstrates how a spider code looks like:

```
import scrapy


class firstSpider(scrapy.Spider):
    name = "first"
    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] + '.html'
        with open(filename, 'wb') as f:
            f.write(response.body)
```

18. Scrapy – Crawling

Description

To execute your spider, run the following command within your *first_scrapy* directory:

```
scrapy crawl first
```

Where, **first** is the name of the spider specified while creating the spider.

Once the spider crawls, you can see the following output:

```
2016-08-09 18:13:07-0400 [scrapy] INFO: Scrapy started (bot: tutorial)

2016-08-09 18:13:07-0400 [scrapy] INFO: Optional features available: ...

2016-08-09 18:13:07-0400 [scrapy] INFO: Overridden settings: {}

2016-08-09 18:13:07-0400 [scrapy] INFO: Enabled extensions: ...

2016-08-09 18:13:07-0400 [scrapy] INFO: Enabled downloader middlewares: ...

2016-08-09 18:13:07-0400 [scrapy] INFO: Enabled spider middlewares: ...

2016-08-09 18:13:07-0400 [scrapy] INFO: Enabled item pipelines: ...

2016-08-09 18:13:07-0400 [scrapy] INFO: Spider opened

2016-08-09 18:13:08-0400 [scrapy] DEBUG: Crawled (200) <GET
http://www.dmoz.org/Computers/Programming/Languages/Python/Resources/>
(referer: None)

2016-08-09 18:13:09-0400 [scrapy] DEBUG: Crawled (200) <GET
http://www.dmoz.org/Computers/Programming/Languages/Python/Books/> (referer:
None)

2016-08-09 18:13:09-0400 [scrapy] INFO: Closing spider (finished)
```

As you can see in the output, for each URL there is a log line which *(referer: None)* states that the URLs are start URLs and they have no referrers. Next, you should see two new files named *Books.html* and *Resources.html* are created in your *first_scrapy* directory.

19. Scrapy – Extracting Items

Description

For extracting data from web pages, Scrapy uses a technique called selectors based on XPath and CSS expressions. Following are some examples of XPath expressions:

- **/html/head/title**: This will select the <title> element, inside the <head> element of an HTML document.

- **/html/head/title/text()**: This will select the text within the same <title> element.

- **//td**: This will select all the elements from<td>.

- **//div[@class="slice"]**: This will select all the elements from *div* which contain an attribute class="slice".

Selectors have four basic methods as shown in the following table:

Sr. No.	Method & Description
1	**extract()** It returns a unicode string along with the selected data.
2	**re()** It returns a list of unicode strings, extracted when the regular expression was given as argument.
3	**xpath()** It returns a list of selectors, which represents the nodes selected by the xpath expression given as an argument.
4	**css()** It returns a list of selectors, which represents the nodes selected by the CSS expression given as an argument.

Using Selectors in the Shell

To demonstrate the selectors with the built-in Scrapy shell, you need to have IPython installed in your system. The important thing here is, the URLs should be included within the quotes while running Scrapy; otherwise the URLs with '**&**' characters won't work. You can start a shell by using the following command in the project's top level directory:

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

A shell will look like the following:

```
[ ... Scrapy log here ... ]
2014-01-23 17:11:42-0400 [scrapy] DEBUG: Crawled (200) <GET
http://www.dmoz.org/Computers/Programming/Languages/Python/Books/>(referer: None)
[s] Available Scrapy objects:
[s]   crawler    <scrapy.crawler.Crawler object at 0x3636b50>
[s]   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]   settings   <scrapy.settings.Settings object at 0x3fadc50>
[s]   spider     <Spider 'default' at 0x3cebf50>
[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
In [1]:
```

When shell loads, you can access the body or header by using *response.body* and *response.header* respectively. Similarly, you can run queries on the response using *response.selector.xpath() or response.selector.css()*.

For instance:

```
In [1]: response.xpath('//title')
Out[1]: [<Selector xpath='//title' data=u'<title>My Book - Scrapy'>]

In [2]: response.xpath('//title').extract()
Out[2]: [u'<title>My Book - Scrapy: Index: Chapters</title>']

In [3]: response.xpath('//title/text()')
Out[3]: [<Selector xpath='//title/text()' data=u'My Book - Scrapy: Index:'>]

In [4]: response.xpath('//title/text()').extract()
Out[4]: [u'My Book - Scrapy: Index: Chapters']

In [5]: response.xpath('//title/text()').re('(\w+):')
Out[5]: [u'Scrapy', u'Index', u'Chapters']
```

Extracting the Data

To extract data from a normal HTML site, we have to inspect the source code of the site to get XPaths. After inspecting, you can see that the data will be in the **ul** tag. Select the elements within **li** tag.

The following lines of code shows extraction of different types of data:

For selecting data within li tag:

```
response.xpath('//ul/li')
```

For selecting descriptions:

```
response.xpath('//ul/li/text()').extract()
```

For selecting site titles:

```
response.xpath('//ul/li/a/text()').extract()
```

For selecting site links:

```
response.xpath('//ul/li/a/@href').extract()
```

The following code demonstrates the use of above extractors:

```
import scrapy

class MyprojectSpider(scrapy.Spider):
    name = "project"
    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):
        for sel in response.xpath('//ul/li'):
            title = sel.xpath('a/text()').extract()
            link = sel.xpath('a/@href').extract()
            desc = sel.xpath('text()').extract()
            print title, link, desc
```

20. Scrapy – Using an Item

Description

Item objects are the regular dicts of Python. We can use the following syntax to access the attributes of the class:

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

Add the above code to the following example:

```
import scrapy

from tutorial.items import DmozItem

class MyprojectSpider(scrapy.Spider):
    name = "project"
    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):
        for sel in response.xpath('//ul/li'):
            item = DmozItem()
            item['title'] = sel.xpath('a/text()').extract()
            item['link'] = sel.xpath('a/@href').extract()
            item['desc'] = sel.xpath('text()').extract()
            yield item
```

The output of the above spider will be:

```
[scrapy] 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 for feedback. [author website, Gnosis Software,
Inc.\n],

      'link': [u'http://gnosis.cx/TPiP/'],

      'title': [u'Text Processing in Python']}

[scrapy] 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 build XML applications fast, Python tutorial, DOM and
SAX, new Pyxie open source XML processing library. [Prentice Hall PTR]\n'],

      'link': [u'http://www.informit.com/store/product.aspx?isbn=0130211192'],

      'title': [u'XML Processing with Python']}
```

21. Scrapy – Following Links

Description

In this chapter, we'll study how to extract the links of the pages of our interest, follow them and extract data from that page. For this, we need to make the following changes in our previous code shown as follows:

```
import scrapy

from tutorial.items import DmozItem

class MyprojectSpider(scrapy.Spider):
    name = "project"
    allowed_domains = ["dmoz.org"]
    start_urls = [
        "http://www.dmoz.org/Computers/Programming/Languages/Python/",
    ]

    def parse(self, response):
        for href in response.css("ul.directory.dir-col > li > a::attr('href')"):
            url = response.urljoin(href.extract())
            yield scrapy.Request(url, callback=self.parse_dir_contents)

    def parse_dir_contents(self, response):
        for sel in response.xpath('//ul/li'):
            item = DmozItem()
            item['title'] = sel.xpath('a/text()').extract()
            item['link'] = sel.xpath('a/@href').extract()
            item['desc'] = sel.xpath('text()').extract()
            yield item
```

The above code contains the following methods:

- **parse()**: It will extract the links of our interest.
- **response.urljoin**: The parse() method will use this method to build a new url and provide a new request, which will be sent later to callback.
- **parse_dir_contents()**: This is a callback which will actually scrape the data of interest.

Here, Scrapy uses a callback mechanism to follow links. Using this mechanism, the bigger crawler can be designed and can follow links of interest to scrape the desired data from different pages. The regular method will be callback method, which will extract the items, look for links to follow the next page, and then provide a request for the same callback.

The following example produces a loop, which will follow the links to the next page.

```
def parse_articles_follow_next_page(self, response):
    for article in response.xpath("//article"):
        item = ArticleItem()

        ... extract article data here

        yield item

    next_page = response.css("ul.navigation > li.next-page > a::attr('href')")
    if next_page:
        url = response.urljoin(next_page[0].extract())
        yield scrapy.Request(url, self.parse_articles_follow_next_page)
```

22. Scrapy – Scraped Data

Description

The best way to store scraped data is by using Feed exports, which makes sure that data is being stored properly using multiple serialization formats. JSON, JSON lines, CSV, XML are the formats supported readily in serialization formats. The data can be stored with the following command:

```
scrapy crawl dmoz -o data.json
```

This command will create a **data.json** file containing scraped data in JSON. This technique holds good for small amount of data. If large amount of data has to be handled, then we can use Item Pipeline. Just like data.json file, a reserved file is set up when the project is created in **tutorial/pipelines.py**.

Scrapy Built-In Services

23. Scrapy – Logging

Description

Logging means tracking of events, which uses built-in logging system and defines functions and classes to implement applications and libraries. Logging is a ready-to-use material, which can work with Scrapy settings listed in Logging settings.

Scrapy will set some default settings and handle those settings with the help of scrapy.utils.log.configure_logging() when running commands.

Log levels

In Python, there are five different levels of severity on a log message. The following list shows the standard log messages in an ascending order:

- **logging.DEBUG** - for debugging messages (lowest severity)
- **logging.INFO** - for informational messages
- **logging.WARNING** - for warning messages
- **logging.ERROR** - for regular errors
- **logging.CRITICAL** - for critical errors (highest severity)

How to Log Messages

The following code shows logging a message using **logging.info** level.

```
import logging
logging.info("This is an information")
```

The above logging message can be passed as an argument using **logging.log** shown as follows:

```
import logging
logging.log(logging.INFO, "This is an information")
```

Now, you can also use loggers to enclose the message using the logging helpers logging to get the logging message clearly shown as follows:

```
import logging
logger = logging.getLogger()
logger.info("This is an information")
```

There can be multiple loggers and those can be accessed by getting their names with the use of **logging.getLogger** function shown as follows.

```
import logging
logger = logging.getLogger('mycustomlogger')
logger.info("This is an information")
```

A customized logger can be used for any module using the *__name__* variable which contains the module path shown as follows:

```
import logging
logger = logging.getLogger(__name__)
logger.info("This is an information")
```

Logging from Spiders

Every spider instance has a **logger** within it and can used as follows:

```
import scrapy
class LogSpider(scrapy.Spider):

    name = 'logspider'
    start_urls = ['http://dmoz.com']

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

In the above code, the logger is created using the Spider's name, but you can use any customized logger provided by Python as shown in the following code:

```
import logging
import scrapy

logger = logging.getLogger('customizedlogger')
class LogSpider(scrapy.Spider):

    name = 'logspider'
    start_urls = ['http://dmoz.com']
    def parse(self, response):
        logger.info('Parse function called on %s', response.url)
```

Logging Configuration

Loggers are not able to display messages sent by them on their own. So they require "handlers" for displaying those messages and handlers will be redirecting these messages to their respective destinations such as files, emails, and standard output.

Depending on the following settings, Scrapy will configure the handler for logger.

Logging Settings

The following settings are used to configure the logging:

- The **LOG_FILE** and **LOG_ENABLED** decide the destination for log messages.
- When you set the **LOG_ENCODING** to false, it won't display the log output messages.
- The **LOG_LEVEL** will determine the severity order of the message; those messages with less severity will be filtered out.
- The **LOG_FORMAT** and **LOG_DATEFORMAT** are used to specify the layouts for all messages.
- When you set the **LOG_STDOUT** to true, all the standard output and error messages of your process will be redirected to log.

Command-line Options

Scrapy settings can be overridden by passing command-line arguments as shown in the following table:

Sr. No.	Command	Description
1	--logfile FILE	Overrides *LOG_FILE*
2	--loglevel/-L LEVEL	Overrides *LOG_LEVEL*
3	--nolog	Sets *LOG_ENABLED* to *False*

scrapy.utils.log module

This function can be used to initialize logging defaults for Scrapy.

```
scrapy.utils.log.configure_logging(settings=None, install_root_handler=True)
```

Sr. No.	Parameters	Description
1	settings (dict, None)	It creates and configures the handler for root logger. By default, it is *None*.
2	install_root_handler (bool)	It specifies to install root logging handler. By default, it is *True.*

The above function:

- Routes warnings and twisted loggings through Python standard logging.
- Assigns DEBUG to Scrapy and ERROR level to Twisted loggers.
- Routes stdout to log, if LOG_STDOUT setting is true.

Default options can be overridden using the **settings** argument. When settings are not specified, then defaults are used. The handler can be created for root logger, when install_root_handler is set to true. If it is set to false, then there will not be any log output set. When using Scrapy commands, the configure_logging will be called automatically and it can run explicitly, while running the custom scripts.

To configure logging's output manually, you can use **logging.basicConfig()** shown as follows:

```
import logging
from scrapy.utils.log import configure_logging


configure_logging(install_root_handler=False)
logging.basicConfig(
    filename='logging.txt',
    format='%(levelname)s: %(your_message)s',
    level=logging.INFO
)
```

24. Scrapy – Stats Collection

Description

Stats Collector is a facility provided by Scrapy to collect the stats in the form of key/values and it is accessed using the Crawler API (Crawler provides access to all Scrapy core components). The stats collector provides one stats table per spider in which the stats collector opens automatically when spider is opening and closes the stats collector when spider is closed.

Common Stats Collector Uses

The following code accesses the stats collector using **stats** attribute.

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

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

The following table shows various options can be used with stats collector:

Sr. No.	Parameters	Description
1	`stats.set_value('hostname', socket.gethostname())`	It is used to set the stats value.
2	`stats.inc_value('customized_count')`	It increments the stat value.
3	`stats.max_value('max_items_scraped', value)`	You can set the stat value, only if greater than previous value.
4	`stats.min_value('min_free_memory_percent', value)`	You can set the stat value, only if lower than previous value.
5	`stats.get_value('customized_count')`	It fetches the stat value.
6	`stats.get_stats()` `{'custom_count': 1, 'start_time': datetime.datetime(2009, 7, 14, 21, 47, 28, 977139)}`	It fetches all the stats.

Available Stats Collectors

Scrapy provides different types of stats collector which can be accessed using the *STATS_CLASS* setting.

MemoryStatsCollector

It is the default Stats collector that maintains the stats of every spider which was used for scraping and the data will be stored in the memory.

```
class scrapy.statscollectors.MemoryStatsCollector
```

DummyStatsCollector

This stats collector is very efficient which does nothing. This can be set using the STATS_CLASS setting and can be used to disable the stats collection in order to improve the performance.

```
class scrapy.statscollectors.DummyStatsCollector
```

25. Scrapy – Sending an E-mail

Description

Scrapy can send e-mails using its own facility called as Twisted non-blocking IO which keeps away from non-blocking IO of the crawler. You can configure the few settings of sending emails and provide simple API for sending attachments.

There are two ways to instantiate the MailSender as shown in the following table:

Sr. No.	Parameters	Method

1	from scrapy.mail import MailSender
	mailer = MailSender()
		By using a standard constructor.

2	mailer = MailSender.from_settings(settings)	By using Scrapy settings object.

The following line sends an e-mail without attachments:

```
mailer.send(to=["receiver@example.com"], subject=" subject data", body="body
data", cc=["list@example.com"])
```

MailSender Class Reference

The MailSender class uses Twisted non-blocking IO for sending e-mails from Scrapy.

```
class scrapy.mail.MailSender(smtphost=None, mailfrom=None, smtpuser=None,
smtppass=None, smtpport=None)
```

The following table shows the parameters used in *MailSender* class:

Sr. No.	Parameters	Description
1	smtphost (str)	The SMTP host is used for sending the emails. If not, then *MAIL_HOST* setting will be used.
2	mailfrom (str)	The address of receiver is used to send the emails. If not, then *MAIL_FROM* setting will be used.
3	smtpuser	It specifies the SMTP user. If it is not used, then *MAIL_USER* setting will be used and there will be no SMTP validation, if it is not mentioned.
4	smtppass (str)	It specifies the SMTP pass for validation.
5	smtpport (int)	It specifies the SMTP port for connection.

6	smtptls (boolean)	It implements using the SMTP STARTTLS.
7	smtpssl (boolean)	It administers using a safe SSL connection.

Following two methods are there in the MailSender class reference as specified. First method,

```
classmethod from_settings(settings)
```

It incorporates by using the Scrapy settings object. It contains the following parameter:

settings (scrapy.settings.Settings object): It is treated as e-mail receiver.

Another method,

```
send(to, subject, body, cc=None, attachs=(), mimetype='text/plain',
charset=None)
```

The following table contains the parameters of the above method:

Sr. No.	Parameters	Description
1	to (list)	It refers to the email receiver.
2	subject (str)	It specifies the subject of the email.
3	cc (list)	It refers to the list of receivers.
4	body (str)	It refers to email body data.
5	attachs (iterable)	It refers to the email's attachment, mimetype of the attachment and name of the attachment.
6	mimetype (str)	It represents the MIME type of the e-mail.
7	charset (str)	It specifies the character encoding used for email contents.

Mail Settings

The following settings ensure that without writing any code, we can configure an e-mail using the MailSender class in the project.

Sr. No.	Settings & Description	Default Value
1	**MAIL_FROM** It refers to sender email for sending emails.	'scrapy@localhost'
2	**MAIL_HOST** It refers to SMTP host used for sending emails.	'localhost'
3	**MAIL_PORT** It specifies SMTP port to be used for sending emails.	25
4	**MAIL_USER** It refers to SMTP validation. There will be no validation, if this setting is set to disable.	None
5	**MAIL_PASS** It provides the password used for SMTP validation.	None
6	**MAIL_TLS** It provides the method of upgrading an insecure connection to a secure connection using SSL/TLS.	False
7	**MAIL_SSL** It implements the connection using a SSL encrypted connection.	False

26. Scrapy – Telnet Console

Description

Telnet console is a Python shell which runs inside Scrapy process and is used for inspecting and controlling a Scrapy running process.

Access Telnet Console

The telnet console can be accessed using the following command:

```
telnet localhost 6023
```

Basically, telnet console is listed in TCP port, which is described in **TELNETCONSOLE_PORT** settings.

Variables

Some of the default variables given in the following table are used as shortcuts:

Sr. No.	Shortcut & Description
1	**crawler** This refers to the Scrapy Crawler (scrapy.crawler.Crawler) object.
2	**engine** This refers to Crawler.engine attribute.
3	**spider** This refers to the spider which is active.
4	**slot** This refers to the engine slot.
5	**extensions** This refers to the Extension Manager (Crawler.extensions) attribute.
6	**stats** This refers to the Stats Collector (Crawler.stats) attribute.
7	**setting** This refers to the Scrapy settings object (Crawler.settings) attribute.
8	**est** This refers to print a report of the engine status.
9	**prefs** This refers to the memory for debugging.

10	**p** This refers to a shortcut to the **pprint.pprint** function.
11	**hpy** This refers to memory debugging.

Examples

Following are some examples illustrated using Telnet Console.

Pause, Resume and Stop the Scrapy Engine

To pause Scrapy engine, use the following command:

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

To resume Scrapy engine, use the following command:

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

To stop Scrapy engine, use the following command:

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

View Engine Status

Telnet console uses **est()** method to check the status of Scrapy engine as shown in the following code:

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

time()-engine.start_time                        : 8.62972998619
engine.has_capacity()                           : False
len(engine.downloader.active)                   : 16
engine.scraper.is_idle()                        : False
engine.spider.name                              : followall
engine.spider_is_idle(engine.spider)          : False
engine.slot.closing                           : False
len(engine.slot.inprogress)                   : 16
len(engine.slot.scheduler.dqs or [])          : 0
len(engine.slot.scheduler.mqs)                : 92
len(engine.scraper.slot.queue)                : 0
len(engine.scraper.slot.active)               : 0
engine.scraper.slot.active_size               : 0
engine.scraper.slot.itemproc_size             : 0
engine.scraper.slot.needs_backout()           : False
```

Telnet Console Signals

You can use the telnet console signals to add, update, or delete the variables in the telnet local namespace. To perform this action, you need to add the telnet_vars dict in your handler.

```
scrapy.extensions.telnet.update_telnet_vars(telnet_vars)
```

Parameters:

```
telnet_vars (dict)
```

Where, dict is a dictionary containing telnet variables.

Telnet Settings

The following table shows the settings that control the behavior of Telnet Console:

Sr. No.	Settings & Description	Default Value
1	**TELNETCONSOLE_PORT** This refers to port range for telnet console. If it is set to none, then the port will be dynamically assigned.	[6023, 6073]
2	**TELNETCONSOLE_HOST** This refers to the interface on which the telnet console should listen.	'127.0.0.1'

27. Scrapy – Web Services

Description

A running Scrapy web crawler can be controlled via **JSON-RPC**. It is enabled by JSONRPC_ENABLED setting. This service provides access to the main crawler object via JSON-RPC 2.0 protocol. The endpoint for accessing the crawler object is:

```
http://localhost:6080/crawler
```

The following table contains some of the settings which show the behavior of web service:

Sr. No.	Setting & Description	Default Value
1	**JSONRPC_ENABLED** This refers to the boolean, which decides the web service along with its extension will be enabled or not.	True
2	**JSONRPC_LOGFILE** This refers to the file used for logging HTTP requests made to the web service. If it is not set the standard Scrapy log will be used.	None
3	**JSONRPC_PORT** This refers to the port range for the web service. If it is set to none, then the port will be dynamically assigned.	[6080, 7030]
4	**JSONRPC_HOST** This refers to the interface the web service should listen on.	'127.0.0.1'