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...
```

End of ebook preview

If you liked what you saw…

Buy it from our store @ **https://store.tutorialspoint.com**