Topics

Monday, April 14, 2025 6:13 AM

1. Introduction to Scrapy
2. Understanding Spiders
3. Working with Scrapy and Scrapy Shell
4. Advanced Features
5. Mini Project
6. Best Practices

1. Introduction

Monday, April 14, 2025 6:17 AM

What is Scrapy?

- Scrapy is an open-source web scraping and web crawling framework written in Python
- It is widely used for extracting structured data from websites in an efficient and scalable way
- Designed for large-scale scraping, data extraction from multiple pages and asynchronous processing

2. Advantages

Monday, April 14, 2025 6:22 AM

- Comes with a project structure, command-line tools, pipelines, etc.
- Asynchronous: Extremely fast and efficient compared to synchronous scrapers like BeautifulSoup and Requests
- Makes it easy to crawl multiple pages and navigate through websites
- Built-in data export: Export to JSON, CSV, XML, or directly to databases
- Reusability: Users can organize and reuse spiders and pipelines easily
- Provides User-Agent Rotation and Proxy Support to help avoid getting blocked

3. Comparison

Monday, April 14, 2025 6:22 AM

Feature	Scrapy	BeautifulSoup	Selenium
Speed	Asynchronous (very fast)	❌ Synchronous	❌ Very slow (renders JS)
Project Structure	✅ Organized	❌ Script-based	❌ Script-based
Best for	✅ Large-scale scraping	✅ Small, simple sites	✅ JS-heavy sites
Built-in pipeline	✅ Yes	❌ No	❌ No
Request Handling	✅ Built-in queueing and retry	❌ Manual	❌ Manual

4. Installation

Monday, April 14, 2025 6:34 AM

1. Create and activate virtual environment:
    - venv
    - conda
2. Use Python version 3.6+
3. Install and verify:
    - Run the command pip install scrapy
    - Check installed version
    - Run the command scrapy

5. Basic Set-up

Monday, April 14, 2025 6:45 AM

1. Navigate to directory of choice

2. Run the command scrapy startproject <project_name>
   - This will create a folder called project_name

3. The below project structure will be generated:

```
myproject/
├── scrapy.cfg
├── myproject/
│   ├── __init__.py
│   ├── items.py
│   ├── middlewares.py
│   ├── pipelines.py
│   ├── settings.py
│   └── spiders/
│       └── __init__.py
```

6. Project Structure

Monday, April 14, 2025 7:02 AM

Scrapy will generate the below project structure by default:

```
myproject/
├── scrapy.cfg
├── myproject/
│   ├── __init__.py
│   ├── items.py
│   ├── middlewares.py
│   ├── pipelines.py
│   ├── settings.py
│   └── spiders/
│       └── __init__.py
```

Purpose of this structure:

- Enhances project scalability
- Improves maintainability
- Components are reusable
- Ready for production and deployment

SCENARIO	HOW PROJECT STRUCTURE HELPS
Adding a new website to scrape	Just add a new file in spiders/
Changing export format	Tweak settings.py
Validating data	Use pipelines.py
Dealing with captchas or blocking	Use middlewares.py
Testing a single spider	Run only that spider with scrapy crawl
Reusing data structure	Define once in items.py and use across spiders

Role of each file in the structure:

FILE	PURPOSE
scrapy.cfg	Project configuration (mainly for deployment or multi-project use)
items.py	Define data structure of scraped items
middlewares.py	Modify requests/responses globally
pipelines.py	Clean/process/store scraped data
settings.py	Central configuration: delays, retries, pipelines, etc.
spiders/	Contains the web scrapers (spiders)
__init__.py	Marks folders as Python packages

7. Running a Spider

Monday, April 14, 2025 7:02 AM

1. **Navigate into the project directory**

2. **Create a spider with the command** --> scrapy genspider <spider_name> <url to scrape>
   - This will create a file myproject/spiders/<spider_name>.py

3. **Edit the script** <spider_name>.py

4. **Run the spider with the command** --> scrapy crawl <spider_name>

5. **To store output in a separate file (ex: JSON)**:
   - Run the command --> scrapy crawl <spider_name>-o output.json
   - This will store the output in the file output.json

1. Agenda

Monday, April 14, 2025 8:52 AM

1. **Definition**
2. **Spider in Scrapy**
3. **Anatomy of a Spider**
4. **Types of Spiders**

2. Definition

Tuesday, April 15, 2025 8:23 AM

In web scraping, a spider (also called a crawler/bot) is a program or automated script that navigates through web pages, extracts information, and often follows links to continue the process on other pages

Basic Workflow of a Spider:

1. Starts at a given URL
2. Downloads the page content (HTML)
3. Extracts necessary data
4. Finds links on the page and follows them if needed
5. Repeats the process for new pages

Spider vs Scraper:

TERM	MEANING
Spider	The agent that navigates through web pages
Scraper	The tool/function that extracts data from a webpage

3. Spider in Scrapy

Tuesday, April 15, 2025 8:42 AM

- In Scrapy, a spider is a Python class that you define to scrape information from a website (or a group of websites)
- It tells Scrapy what URLs to start with, how to follow links and how to extract the data you need from the pages

Key Points of a Spider in Scrapy:

- A spider inherits from scrapy.Spider
- It must have a unique name
- It defines one or more start URLs
- It includes a parse() method, which handles the response and extracts data or follows links

4. Anatomy

Tuesday, April 15, 2025 8:29 AM

Components of a Spider:

- The **name** attribute:
  - *Every spider must have a unique name attribute*
  - *This name is used to run/execute the spider*

- The **start_urls** attribute:
  - *Defines the initial requests made when the spider starts*
  - *These denote the entry points for crawling*
  - *Scrapy will automatically make a request to each URL in this list and call the parse() method with the response*

- The **parse()** method:
  - *Is the default callback function to handle responses*
  - *It extracts data from the page using selectors like CSS or Xpath*
  - *It can also yield new requests to other pages*

- The **response.follow()** method:
  - *Used to crawl internal links from one page to the next*
  - *It automatically resolves relative URLs*
  - *It's preferred over scrapy.Request() for internal navigation*

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

5. Types

Tuesday, April 15, 2025 8:43 AM

Scrapy offers specialized spider classes for different use-cases:

1. Basic Spider:

- Can be inherited from scrapy.Spider
- Most commonly used
- Allows users to manually control:
  - *Starting URLs*
  - *How pages are parsed*
  - *How links are followed*
- Great for custom logic and flexibility

2. CrawlSpider (Rule-Based Spider):

- Inherits from scrapy.spiders.CrawlSpider
- Uses rules to automate link-following behavior

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

class MyCrawlSpider(CrawlSpider):
    name = 'rule_spider'
    start_urls = ['http://quotes.toscrape.com']

    rules = (
        Rule(LinkExtractor(allow='page/\d+/'), callback='parse_page', follow=True),
    )

    def parse_page(self, response):
        # your logic here
```

3. XMLFeedSpider:

- Used to parse XML feeds (like RSS, sitemap, API responses)
- Handles large feeds efficiently without loading the whole document into memory

```python
from scrapy.spiders import XMLFeedSpider

class MyXMLSpider(XMLFeedSpider):
    name = 'xml_spider'
    start_urls = ['http://example.com/feed.xml']
    itertag = 'item'

    def parse_node(self, response, node):
        yield {
            'title': node.xpath('title/text()').get(),
            'link': node.xpath('link/text()').get(),
        }
```

4. CSVFeedSpider:

- Used to scrape CSV files
- Automatically splits rows and maps them to columns

```python
from scrapy.spiders import CSVFeedSpider

class MyCSVSpider(CSVFeedSpider):
    name = 'csv_spider'
    start_urls = ['http://example.com/data.csv']
    delimiter = ','

    def parse_row(self, response, row):
        yield {
            'name': row['Name'],
            'price': row['Price'],
        }
```

When to use which Spider?

SPIDER TYPE	USAGE
Basic Spider	You need full control and custom logic for crawling and parsing
CrawlSpider	You want to follow links using URL patterns (e.g. pagination, category pages)
XMLFeedSpider	You're scraping structured XML data (sitemaps, RSS feeds, APIs)
CSVFeedSpider	You're scraping data from CSV sources (e.g. CSV download links)

1. Agenda

Monday, April 14, 2025 8:28 AM

1. Scrape the website: https://quotes.toscrape.com
2. Explore Scrapy Shell
3. Implement Scrapy Spider via Python Script

2. Project Workflow

Monday, April 14, 2025 8:41 AM

1. Set-up project structure
2. Set-up appropriate scripts
3. Explore the response object
4. View page content
5. Extract the page title
6. Extract all quotes from a page
7. Extract all author names
8. Extract tags for the first quote
9. Navigate to the next page link

3. Scrapy Shell

Monday, April 14, 2025 8:39 AM

1. **Open the Scrapy shell instance:**
   - scrapy shell "http://quotes.toscrape.com"

2. **Explore the response object:**
   - response
   - response.status
   - response.headers
   - print(response.text[:500])

3. **View page content:**
   - To view structured tags: response.css('div.quote')[:1]
   - To preview HTML: response.css('div.quote').get()

4. **Extract the page title:**
   - Using CSS: response.css('title::text').get()
   - Using XPATH: response.xpath('//title/text()').get()

5. **Extract all quotes from a page:**
   - response.css('div.quote span.text::text').getall()

6. **Extract all author names:**
   - response.css('small.author::text').getall()

7. Extract tags for the first quote:

- first_quote = response.css('div.quote')[0]
- first_quote.css('div.tags a.tag::text').getall()

8. Navigate to the next page link:

- next_page = response.css('li.next a::attr(href)').get()
  - *This will extract the value of the href attribute*
  - *This returns a relative URL*
- next_page_url = response.urljoin(next_page)
  - *This will return the absolute URL of the next page*
  - *Combines the base URL and relative URL of the next page*
- fetch(next_page_url)
  - *This command is specific to scrapy shell*
  - It will:
    - *Send a request to the URL*
    - *Update the response object with the HTML of the next page*
    - *Allow to manually test selectors on the new page*

4. Scrapy Spider

Monday, April 14, 2025 8:40 AM

1. Agenda

Wednesday, April 16, 2025 6:59 AM

- **Custom Spider Settings**
- **Item Pipelines**
- **User-Agent Rotation & Proxy Usage**
- **Working with Login Pages**
- **Handling APIs**

2. Custom Spider Settings

Saturday, April 19, 2025 9:46 AM

When to Use?

- Override Scrapy's global settings (like download delay, concurrency, export format, etc.) only for a specific spider
- For different spiders to behave differently (e.g., one saves to JSON, another to CSV)
- You want to scrape slowly to avoid detection

CUSTOM SETTING	PURPOSE
DOWNLOAD_DELAY	Waits between each request to reduce load on the server
CONCURRENT_REQUESTS = 1	Ensures only one request is made at a time
FEED_URI	Specifies the output file path
FEED_FORMAT = json	Saves scraped data as JSON
FEED_EXPORT_ENCODING	Makes sure special characters are saved correctly (like emojis, etc.)
LOG_LEVEL	Reduces noise in logs, shows only important messages

3. Item Pipelines

Saturday, April 19, 2025 10:04 AM

When to Use?

- Cleaning text (removing whitespace, quotes)
- Validating fields
- Inserting data into a database (ex: MongoDB, PostgreSQL)

Files to Update:

FILE	PURPOSE	USAGE
items.py	Defines structure for scraped data	At import time and when items are yielded
settings.py	Configures your project (pipelines, user-agent, etc.)	At project startup (e.g., scrapy crawl)
pipelines.py	Contains logic to process and store data	For every item yielded

items.py:

- Make use of Item and Field
- Usage of **Item**:
  - *Defines a clear structure for the scraped data*
  - *Acts as a blueprint or schema for the data*
  - *Enables serialization the data for exporting in formats like JSON, CSV, or XML*
- Usage of **Field**:
  - *Defines the individual attributes of an Item*
  - *Fields can be customized with default values, serializers, and other metadata*
  - *Customize how individual fields behave, such as making them required, setting default values, or specifying validators*
  - *Apply custom transformations to the data in Field() using serializers*
- Together, Item and Field enable better data management, validation, transformation, and integration with other Scrapy components like pipelines and exporters

settings.py:

- It holds global settings for the project, which will be used by all spiders within the it
- Helps centralize the configuration in one place, making it easier to manage and update
- Adjust settings based on crawling needs, such as throttling requests or enabling/disabling middlewares
- Manage how items are processed after they are scraped
- Define which pipeline to use and set the order of execution

pipelines.py:

- Commonly used for data cleaning or applying transformations to ensure that the scraped data is valid and consistent
- Identify and remove duplicate items
- Log items or errors, which is useful for debugging or monitoring the scraping process
- Store the scraped data in different storage systems such as databases, files or cloud

4. User-Agent Rotation & Proxy Usage

Saturday, April 19, 2025 11:50 AM

1. **User-Agent Rotation**:
    - Involves changing the User-Agent header for each request sent
    - The User-Agent header tells the server what browser (and version) is making the request
    - By rotating the User-Agent, you can simulate requests coming from different browsers and devices, helping avoid detection from anti-bot mechanisms that block or rate-limit traffic based on consistent User-Agent headers

2. **Proxy Usage**:
    - Proxies are servers that act as intermediaries between the scraper and the website to scrape
    - Using proxies, you can route your requests through different IP addresses, allowing you to bypass IP-based blocking mechanisms
    - Rotating proxies can be used to make requests from different locations or at different times to avoid detection

Steps:

- Create Scrapy Project
- Edit items.py
- Create and edit a spider script
- Create custom middleware for User-Agent Rotation & Proxy Usage
- Add middleware to settings.py

How it Works:

- Scrapy will call the User-Agent middleware and attach a random User-Agent header to the request:
    - It will call the process_request method of this class for each request Scrapy makes
    - The random user-agent is assigned by request.headers['User-Agent'] = user_agent

- Scrapy will then call the Proxy middleware:
    - Will assign a random proxy to the request using request.meta['proxy']
    - This causes Scrapy to route the request through the selected proxy

- The request will then be sent to the website with a different User-Agent and from a different proxy, helping prevent the server from blocking your scraper

- In the **settings.py** file, we use DOWNLOADER_MIDDLEWARES to enable the middlewares:
    - The keys should be the full path to the middleware class (e.g., 'quotes_scraper.middlewares.UserAgentMiddleware') and the values are the priority number
    - The lower the number, the earlier the middleware is applied

5. Working with Login Pages

Saturday, April 19, 2025 12:59 PM

Aim:

- Visit https://quotes.toscrape.com/login
- Login successfully
- Scrape data after login

Flow of Control:

- Visit the URL mentioned in start_urls
- parse() is automatically called with the response:
  - It submits the login form with dummy credentials
  - Uses FormRequest.from_response() to automatically fill the form fields and submit them
  - It then sends the response to after_login() method for post-login logic
- after_login() is called:
  - Verifies login success by checking if the page contains Logout
  - If login failed, logs an error and stops
  - If login is successful:
    - Sends a new request to the homepage, where quotes are displayed
    - The callback=self.parse_quotes means the next response goes to parse_quotes()
    - dont_filter=True ensures the homepage is fetched again even though it was seen during login redirects
- When parse_quotes() is called:
  - extracts the actual quote data
  - looks for the next page link and continues scraping by recursively calling itself
  - This is pagination in action in Scrapy

Summary:

METHOD	PURPOSE
start_urls	Defines where Scrapy starts — the login page
parse()	Submits the login form with credentials
after_login()	Checks login success and proceeds to homepage scraping
parse_quotes()	Scrapes quotes and follows pagination

6. Handling APIs

Saturday, April 19, 2025 1:15 PM

Aim:

- Fetch posts from https://jsonplaceholder.typicode.com/posts

Steps:

- Update items.py
- Update posts_scraper.py

Summary:

METHOD	PURPOSE
start_urls	Scrapy starts by sending a GET request to the JSON API
parse()	The response received is JSON which is parsed with .json() method
Loop over	Each post in the JSON object is looped and its fields extracted
yield item	Each post is returned as a structured Scrapy item

1. Agenda

Saturday, April 19, 2025 1:53 PM

- **Scrape the website - https://books.toscrape.com/**
- **Extract details for each book:**
  - Title
  - Price
  - Rating
- **Extract data from multiple pages**
- **Clean the extracted data**
- **Export cleaned data to JSON file**

2. Steps

Sunday, April 20, 2025 9:39 AM

- **Create a project:**
  - scrapy startproject mini_project
  - cd mini_project

- **Define the schema - items.py**
  - Title
  - Price
  - Rating

- **Create and update the spider - books_spider.py**
  - scrapy genspider books_scraper https://books.toscrape.com

- **Perform data cleaning - pipelines.py**
  - Title - remove leading/trailing whitespaces
  - Price -
    - Remove currency symbols
    - Remove leading/trailing whitespaces
    - Convert to float type
  - Rating -
    - Extract the rating value
    - Map to appropriate numeric value

- **Enable the pipeline - settings.py**

- **Run the spider - scrapy crawl books -o result.json**

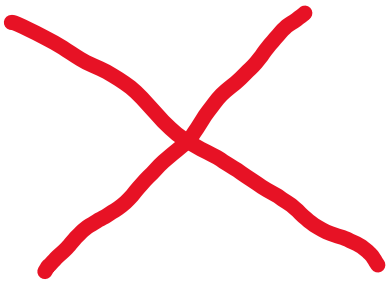
Best Practices

Saturday, April 19, 2025 1:53 PM

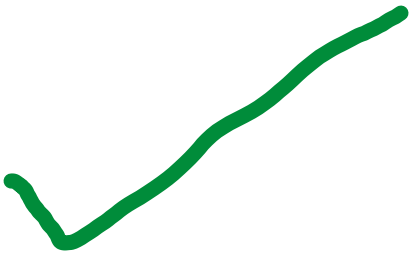
1. Use the Item Class:

- Defined in items.py
- Yield data in the spider using the Item object
- Helps improve the structure, readability, and validation of the data

```
yield {
    'title': title,
    'price': price
}
```



```
item = MyProductItem()
item['title'] = title
item['price'] = price
yield item
```



2. Keep Your Spiders Focused:

- Each spider should scrape one type of page or dataset
- This improves maintainability

3. Use XPath or CSS Selectors Wisely:

- **CSS Selectors** - cleaner and faster for simple HTML
- **XPath** - more powerful for complex nested structures

4. Export Data with FEEDS in settings.py:

- Exports are cleaner this way

```
FEEDS = {
    'output/quotes.csv': {
        'format': 'csv',
        'overwrite': True
    }
}
```

5. **Use Pipelines for Cleaning and Post-Processing:**
    - Update pipelines.py
    - Keep spiders clean and less cluttered

6. **Respect Robots.txt and Site Load**
    - ROBOTSTXT_OBEY = True
    - AUTOTHROTTLE_ENABLED = True

7. **Monitor progress with Logging**

8. **Use Proxies & Rotating User Agents (If Needed)**

9. **Use scrapy shell for Selector Testing**

10. **Use Environment Variables for Secrets:**
    - Never hard-code credentials or API keys
    - Use .env files or Python's os.environ