I

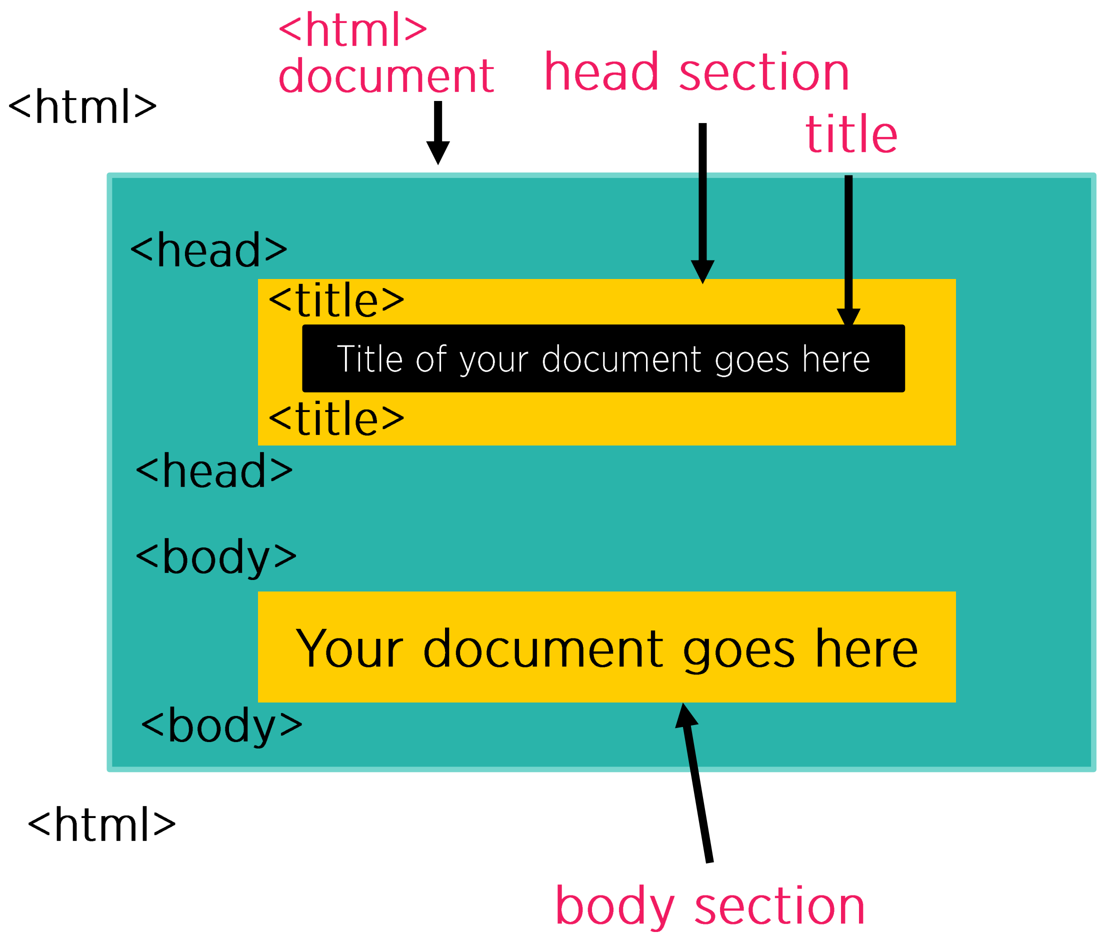
Introduction to Web Scraping

Web scraping

Web scraping is a technique to automatically access and extract large amounts of information from a website, which can save a huge amount of time and effort.

Sample structure of a HTML

- HTML-Stands for Hypertext Markup Language
- Describes the structure of Web pages using markup language
- HTML elements are the building blocks of HTML pages
- HTML elements are represented by HTML tags



HTML tags explained

- The <!DOCTYPE html> declaration defines this document to be HTML5
- The <html> element is the root element of an HTML page
- The <head> element contains title of document
- The <title> element specifies a title for the document
- The <body> element contains the visible page content
- The <h1> element defines a large heading
- The <p> element defines a paragraph
- The <b> element defines the bold of text

HTML headings

- Headings are title of HTML pages
- Headings are defined with the *<h1>* to *<h6>* tags.
- h1 is the largest heading tag while h6 is the smallest heading tag.
- *<h1>* headings should be used for main headings, followed by *<h2>* headings, then the less important *<h3>*, and so on

Paragraphs and breaking lines

Paragraph tag

Most browsers render (process) this with blank lines between each paragraph
<p> text *</p>*

Line break tag

Used when the webmaster wants a carriage return but doesn't want a blank line to follow
*
*

Lists – Unordered Lists

- HTML supplies several list elements. Most list elements are composed of one or more *<li>* (List Item) *</li>* elements.
- UL : Unordered List: Items in this list start with a list mark such as a bullet. Browsers will usually change the list mark in nested lists.

```
 <ul>
        <li>Item One </li>
        <li>Item Two </li>
        <li>Item Three </li>
        <li>Item Four </li>
</ul>
```

Unordered list –

- Item One
- Item Two
- Item Three
- Item Four

Lists – Ordered Lists

- Ordered List: Items in this list are numbered automatically by the browser.

```
<ol>
        <li> Item One </li>
        <li> Item Two </li>
        <li> Item Three </li>
        <li> Item Four </li>
</ol>
```

- You have the choice of setting the TYPE Attribute to one of five numbering styles.

Type	Numbering Styles	
1	Arabic numbers	1,2,3, ……
a	Lower alpha	a, b, c, ……
A	Upper alpha	A, B, C, ……
i	Lower roman	i, ii, iii, ……
I	Upper roman	I, II, III, …….

Lists – Definition Lists

This kind of list is different from the others. Each item in a DL consists of one or more Definition Terms (DT elements), followed by one or more Definition Description (DD elements).

```
<dl>
          <dt>List Name One
                    <dd>This is where information about List Name One would go</dd>
          </dt>
          <dt>List Name Two
                    <dd>This is where information about List Name Two would go</dd>
          </dt>
</dl>
```

Links

- The anchor tag `<a>` is used to link one document to another or from one part of a document to another part of the same document.
- Basic Links:

```
<a href="http://www.stanford.edu/">Stanford University</a>
```

- Inter-document Links:

```
<a href="#spot">Point to 'spot' in this document</a>
```

- Defining a point in a document:

```
<a name="spot">Spot</a>
```

- Email links:

```
<a href="mailto:someone@somehost.com">Email someone@somehost.com</a>
```

HTML images

- Use the HTML <img> element to define an image
- Use the HTML src attribute to define the URL of the image
- Use the HTML alt attribute to define an alternate text for an image, if it cannot be displayed
- Use the HTML width and height attributes to define the size of the image
- Note: Loading images takes time. Large images can slow down your page. Use images carefully.

HTML class & id

- You can create customized classes by adding the class attribute to HTML tags:

- The syntax for creating a class is:
  <h4 class=“class_name” /h4>
  h4 is the HTML header tag
  class_name is the name of the class

- The id attribute must be unique; there can not be more than one tag with the same id value.

- The syntax for creating an id is:
  <h4 id=“id_name” /h4>
  h4 is the HTML header tag
  id_name is an id name assigned to the tag

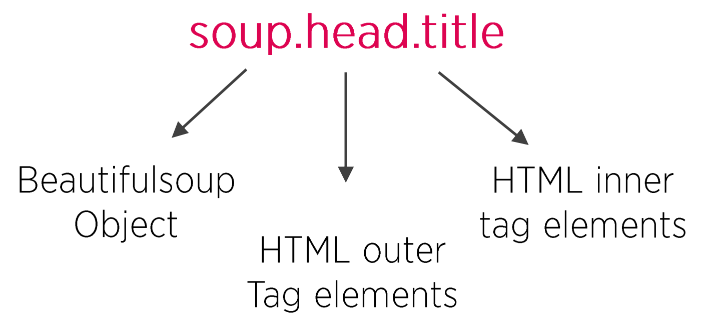
II
Introduction to BeautifulSoup

Web Scraping in Python with BeautifulSoup

- BeautifulSoup is basically an HTML and XML parser and requires additional libraries such as requests, urlib2 to open URLs and store the result
- BeautifulSoup is relatively easy to understand, used for simple programming tasks with efficiency and can get smaller tasks done in no time
- BeautifulSoup is slower than Scrapy if you do not use multiprocessing
- If the project does not require much logic, BeautifulSoup is good for the job, but if you require much customization such as proxies, managing cookies, and data pipelines, Scrapy is the best option.

BeautifulSoup Functions

```
URL = ‘https://en.wikipedia.org/wiki/List_of_game_engines’
        content = requests.get(URL)
        soup = BeautifulSoup(content.text, 'html.parser')
soup.head.title            # returns <title>Head's title</title>
soup.find_all('a')         # if we want all a tags in doc.html
soup.find('a' , href = True)['href']  # returns http://example.com/element1
soup.get_text()
```



BeautifulSoup Example

```
<html>
  <head>
    <title> A simple example page
    </title>
  </head>
 <body>
   <div> <p class="inner-text first-item" id="first"> First paragraph. </p> </div>
      <p class="outer-text first-item" id=”first">
      <b> First outer paragraph. </b>
      </p>
      <p class="outer-text">
      <b> Second outer paragraph. </b> </p>
 </body>
</html>
```

BeautifulSoup Example

```
soup.find_all(id = "first")
```

[<p class="inner-text first-item" id="first"> First paragraph. </p>, <p class="outer-text first-item" id=”first"> <b> First outer paragraph. </b>]

```
soup.find_all('p', class_='outer-text')
```

[<p class="outer-text first-item" id="second"> <b> First outer paragraph. </b> </p>, <p class="outer-text"> <b> Second outer paragraph. </b> </p>]

Using CSS

- BeautifulSoup objects support searching a page via CSS selectors using the select method.

```
soup.select("div p")
```

[<p class="inner-text first-item" id="first"> First paragraph. </p>]

Extracting information

```
page = requests.get("http://forecast.weather.gov/MapClick.php?lat=37.7772&lon=-122.4168")
soup = BeautifulSoup(page.content, 'html.parser’)
seven_day = soup.find(id = "seven-day-forecast")
forecast_items = seven_day.find_all(class_ = "tombstone-container")
tonight = forecast_items[0]
print(tonight.prettify())
```

BeautifulSoup Extracting information

```
<html>
  <head>
    <title>A simple example page
    </title>
  </head>
 <body>
   <div class="tombstone-container">
      <p class="period-name">
      Tonight
      </p>
      <p class="short-desc"> Mostly Clear </p>
      <p class="temp temp-low"> Low: 49 °F </p>
   </div>
  </body>
</html>
```

Extracting information

```
period = tonight.find(class_= 'period-name').get_text()
short_desc = tonight.find(class_= 'short-desc').get_text()
temp = tonight.find(class_= ‘temp').get_text()
print(period)
print(short_desc)
print(temp)


“’result
Tonight
Mostly Clear
Low: 49 °F “’
```

III
Introduction to Scrapy

What is Scrapy?

Scrapy is a Python framework for large scale web scraping. It gives you all the tools you need to efficiently extract data from websites, process them as you want, and store them in your preferred structure and format.

- With Scrapy you can:
  - Fetch millions of data efficiently
  - Run it on server
  - Run spider in multiple processes

Web Scraping in Python with Scrapy

- Scrapy is the complete package for downloading web pages, processing them and save it in files and databases
- Scrapy is a powerhouse for web scraping and offers a lot of ways to scrape a web page. It requires more time to learn and understand how Scrapy works but once learned, eases the process of making web crawlers and running them from just one line of command.
- Scrapy can get big jobs done very easily. It can crawl a group of URLs in no more than a minute depending on the size of the group and does it very smoothly asynchronously (non-blocking) for concurrency.

  Information: Synchronous means that you have to wait for a job to finish to start a new job while Asynchronous means you can move to another job before the previous job has finished

Scrapy Installation

- If you are using Anaconda :
  conda install -c conda-forge scrapy

- Alternatively, use pip for Windows (Linux, Mac) from command prompt:
  pip install scrapy
  or
  python –m pip install scrapy

- For jupyter notebook:
  !pip install scrapy

Scrapy Shell

- Similar to Python shell Scrapy provides a shell of its own that you can use to experiment.

- To start the **scrapy shell** type the following in your command line
  scrapy shell
  (If you are using anaconda, you can write it at the anaconda prompt as well.)

- You can use the Scrapy shell to see what components the web page returns.

Fetch command

- To run Scrapy crawler you have to use **fetch** command in the Scrapy shell:

```
fetch("https://www.reddit.com/r/gameofthrones/")
```

- A crawler or spider goes through a webpage downloading its text and metadata.

- The crawler returns a response which can be viewed by using the view(response) command on shell:

```
view(response)
```

- This command will open the downloaded page in your default browser.

- You can view the raw HTML script by using the following command in Scrapy shell:

```
print(response.text)
```

Using CSS Selectors for Extraction

- You can extract this using the element attributes or the css selector like class, id, etc.

- response.css(..) is a function that helps extract content based on css selector passedto it

  - Let’s say, we have the following <a> tag with a class “product” and the text containing the product name:

```
<a class = “product” href = “reddit.com/…/“ title = “CHUWI Original” target = “blank”>
CHUWI Hi9 Air Tablet PC Android 0
</a>
```

Using CSS Selectors for Extraction

You can extract the text using the element attributes or the css selector like classes:

```
response.css(".product::text").extract_first()
```

extract_first() returns the first element that matches the selector.

"." is used with the product because it's a css . Use *"::"* text to tell your scraper to extract only text content of the matching elements. If you want to extract all the product names use extract():

```
response.css(".product::text").extract()
```

Example

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

Using CSS Selectors for Extraction

To extract the textual data you must use .get() or .getall() methods.
get() – always returns a single result; If there are more than 1 matches then it returns the first one;

If there are no matches it returnsNone get(default='not-found') sets default return value None to "not-found" getall() – returns a list with results:

```
response.css('title::text').get() – Output will be:
                    'Example website'
response.css('img').xpath('@src').getall()
          ['image1_thumb.jpg','i
          mage2_thumb.jpg',
          'image3_thumb.jpg',
          'image4_thumb.jpg',
          'image5_thumb.jpg']
```

Using CSS Selectors for Extraction

extract_first() commonly used in previous Scrapy versions (i.e., alies of .get())

extract() commonly used in previous Scrapy versions (i.e., alies of .getall())
.attrib can be used instead if *@src*; it returns attributes for the first matching element:

1. response.css('img').attrib['src'] => 'image1_thumb.jpg' is equivalent to
2. response.css('img::attr(src)').get() => 'image1_thumb.jpg' is equivalent to
3. response.css('img').xpath('@src').get() => 'image1_thumb.jpg'
4. response.css('base').attrib['href'] => 'http://example.com/' is equivalent to
5. response.css('base::attr(href)').get() => 'http://example.com/' is equivalent to
6. response.css('base').xpath('@href').get() => 'http://example.com/'

Using CSS Selectors for Extraction

```
<html>
  <head>
        <item>
        <title> My Website </title>
      </item>
  </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>
    <dc:creaor><![CDATA[Qss, Bootcamp]]></dc:creator>
  </body>
 </html>
```

Using XPath for Extraction

- XPath (XML Path Language) is a query language for selecting nodes from an XML document. You can navigate through an XML document using XPath.

- Similar to response.css(..) , the function response.xpath(..) in scrapy to deal with Xpath This will show you all the code under the <html> tag:

  response.xpath('/html').extract()

/ means direct child of the node
// means descendent nodes

- If you want to get the <div> tags under the html tag you will write:

  response.xpath('/html//div').extract()

- Behind the scenes, Scrapy uses Xpath to navigate to HTML document items. The CSS selectors you used above are also converted to XPath, but in many cases, CSS is very easy to use

Using XPath for Extraction

- Let’s extract the title of the first post.

```
response.xpath("//item/title").extract_first()
```

```
# result
u'<title> My Website </title>‘
```

Using XPath for Extraction

- To filter out the output to only get the text content of the title:
  - response.xpath("//item/title/text()").extract_first()
- The output:
  - u' My Website'
- Notice that **text()** here is equivalent of **::text** from CSS selectors.
- XPath **//item/title/text()** here you are basically saying find the element "item" and extract the "text" content of its sub element "title".

Using XPath for Extraction

- If you want to get all <div> tags instead of using /html tag you can do:

```
response.xpath("//div").extract()
```

- You can further filter your nodes that you start from and reach your desired nodes by using attributes and their values:

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

- Use /text() to extract all text inside nodes:

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

Using XPath for Extraction

- “creator” tag has some text “dc:” because of which it can’t be extracted using XPath and the author name itself is crowded with “![CDATA..” irrelevant text.
- These are just XML namespaces and so we’ll ask Scrapy to remove the namespace:

```
<dc:creaor><![CDATA[Qss, Bootcamp]]></dc:creator>
```

Dealing with namespaces in XML
Removing namespaces

- Removing namespaces:
  response.selector.remove_namespaces()
- Now when you try extracting the author name , it will work :
  response.xpath("//item/creator/text()").extract_first()
- Output : u' Qss, Bootcamp'

Site for Web Scraping

- For practical purpose, we will go through an easy example of how to automate downloading cars' names from the web site "turbo.az"

- First go to this web-site:
  https://turbo.az

Web Scraping

- Read through the website's Terms and Conditions to understand how you can legally use the data. Most sites prohibit you from using the data for commercial purposes.
- Make sure you are not downloading data at too rapid a rate because this may break the website. You may potentially be blocked from the site as well.

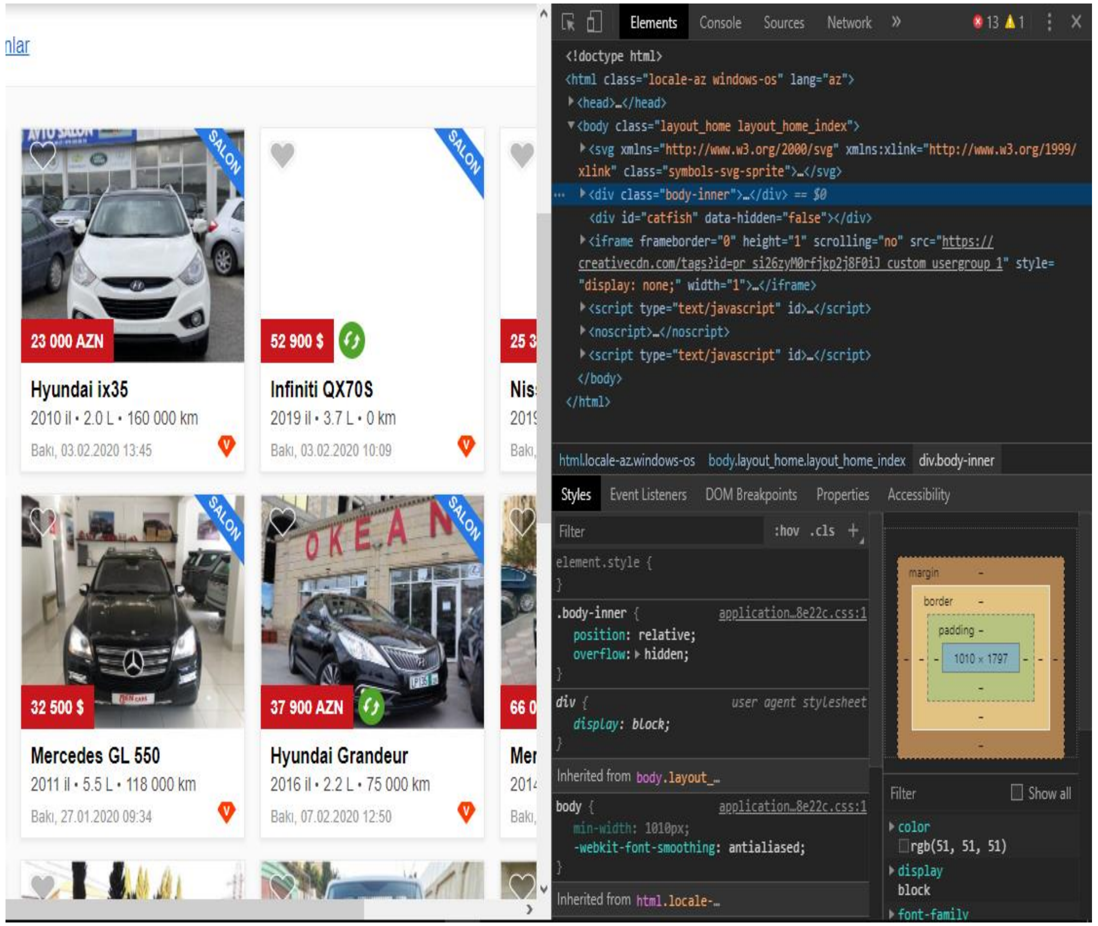
Important libraries:

```
import requests
from bs4 import BeautifulSoup
import pandas as pd
```

Web Scraping

- Go turbo.az website
- Figure out where are our cars' names inside the multiple levels of HTML tags. There is a lot of code on a website page and we want to find the relevant pieces of code that contains our data.
- Right click and click on "Inspect". This allows you to see the raw code behind the site.
- Click on arrow and then click on an area of the site itself, the code for that particular item will be highlighted in the console.
- We will define "https://turbo.az" as url.

url= "https://turbo.az"



```
<!doctype html>
<html class="locale-az windows-os" lang="az">
 ▶<head>…</head>
 ▼<body class="layout_home layout_home_index">
   ▶<svg xmlns="http://www.w3.org/2000/svg" xmlns:xlink="http://www.w3.org/1999/
   xlink" class="symbols-svg-sprite">…</svg>
…  ▶<div class="body-inner">…</div> == $0
    <div id="catfish" data-hidden="false"></div>
   ▶<iframe frameborder="0" height="1" scrolling="no" src="https://
   creativecdn.com/tags?id=pr_si26zyM0rfjkp2j8F0iJ_custom_usergroup_1" style=
   "display: none;" width="1">…</iframe>
   ▶<script type="text/javascript" id>…</script>
   ▶<noscript>…</noscript>
   ▶<script type="text/javascript" id>…</script>
  </body>
</html>
```

html.locale-az.windows-os   body.layout_home.layout_home_index   div.body-inner

Styles   Event Listeners   DOM Breakpoints   Properties   Accessibility

```
Filter                              :hov .cls +
element.style {
}
.body-inner {              application…8e22c.css:1
  position: relative;
  overflow: ▶ hidden;
}
div {                      user agent stylesheet
  display: block;
}
Inherited from body.layout_…
body {                     application…8e22c.css:1
  min-width: 1010px;
  -webkit-font-smoothing: antialiased;
}
Inherited from html.locale-…
```

```
margin -
border -
padding -
1010 × 1797

Filter                    Show all
▶ color
  rgb(51, 51, 51)
▶ display
  block
▶ font-family
```

Web Scraping

- Next we parse the html with BeautifulSoup so that we can work with a nicer, nested BeautifulSoup data structure. If you are interested in learning more about this library, check out the BeatifulSoup documentation.

- Structure of Beautiful Soup

  Beautiful Soup(response.text, “html.parser”)

Code Structure

```
# Set the URL you want to webscrape from
      url = “LINK”

# Connect to the url
     response = requests.get(url)

# Parse HTML and save to BeautifulSoup object
    soup = BeautifulSoup(response.text, “html.parser”)

# To download the whole data set, let's do a loop through all tags
       fin_names = []
       names = car_soup.findAll('p',attrs={"class":"products-name"})
       for car_name in names:
             fin_names.append(car_name.text)
       print(fin_names)

# Download data as csv
      cars = pd.DataFrame(fin_names)
      cars_csv = cars.to_csv('turbo_cars_names.csv')
```

Scrapy

- Installation: *pip install scrapy*
- Documentation link: https://docs.scrapy.org/en/latest/intro/install.html
- Scrapy is an application framework for crawling web sites and extracting structured data which can be used for a wide range of useful applications, like data mining, information processing or historical archival.