Automated data collection

M. Fuat Kına

Data collection

- Sources for readymade data: World Bank, OECD, GLOCON, TÜİK, etc.
- Application programing interfaces: Social media, messengers, etc.
- Web scrapping
- Advanced data extraction techniques: AI, ML, etc.

Application programing interfaces

- Web APIs are provided by website operators
- Access restrictions (authentication, rate limits)

Social Media

- Facebook (https://developers.facebook.com/)
- Twitter (https://developer.twitter.com/en/docs)
- YouTube (https://developers.google.com/youtube/v3)
- Flickr (https://www.flickr.com/services/api/)
- Reddit (https://www.reddit.com/dev/api/)
- LinkedIn (https://www.linkedin.com/developers/)
- **Messenger**: Telegram, WhatsApp, Threema, Skype, Discord
- **Streaming**: Spotify, Apple Music, Vimeo, Twitch
- **Other Services**: Google Maps, Amazon, Wikipedia

Twitter API

- Easy to access
- Rich data content
  - Retweet, favorite, follower counts
  - Tweet content, bio
  - Profile picture
- Academic API provides 10 million tweets per month.

Challenges:

  - It is hard to make sure about real human beings' accounts
  - Lack of information about who those people are
  - Platform affordances

Relevant library in python

tweepy

Practical session for twitter API

Web scrapping

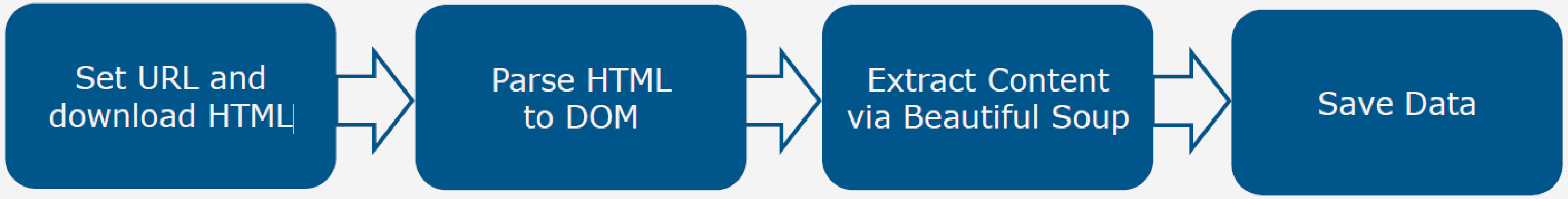
- Inspect the webpage and the HTML code
- Set URL and download HTML
- Cut out HTML
- Extract content
- Save your data

Relevant libraries in python

Scrapy, BeautifulSoup, etc.

Webscraping with Beautiful Soup

- Python library for extracting data from HTML and XML files
- Official documentation: https://www.crummy.com/software/BeautifulSoup/bs4/doc/#



```
requests.get(url)    BeautifulSoup(...)    soup.select('article')    df.to_csv(...)
```

URL and HTML structures

- url = path + “/” + searchterm + “?” + parameters

https://www.aljazeera.com/search/Turkey?sort=date

- html = elements + attributes + text
  - Hypertext markup language
  - Children, parents and siblings, descendants

```
<html>
    <body>
        <div id="div1" class="class-1">
            <p class="class-1 class-2">Hello World!</p>
            <p>Data Science!</p>
        </div>
    </body>
</html>
```

Some HTML-Elements

Metadata:

<head> collection of metadata

<title> title of the document

Grouped Contents

<div> container

<p> paragraph

Lists

<ul> unordered list

<ol> ordered list

<li> entry of a list

Sections:

<body> main contents of the document

<section> section of the document

<h1>, <h2>, … headlines

Links

<a> Hyperlink, refers via href-attribute to resource

Tables

<table> table

<tr> row of a table

<td> cell of a table

XPATHS

to parse an html file with Scrapy

For the rest of the presentation, I would like to give credits for: https://app.datacamp.com/learn/courses/web-scraping-with-python

- / → The single forward-slash moves us forward one generation
- [] → The brackets specify the order of the noted element
- // → The double-forward slash tells us to look forward to all future generations
- *Xpath = '/html/body/div[2]//table'*

meaning that all tables under the second div element, which is child of first body element of the head element.

Examples:

'//p'

'/head/body//div'

'/div[3]//p'

- To select attributes use:
- Xpath = '//span[@class="span-class"]'

meaning that all span elements, whose class attribute is 'span_class'.

XPATHS

Example: Choose Data Science!

```
html =
'''<html>
  <body>
    <div>
      <p>Hello World!</p>
      <div>
        <p>Data Science!</p>
      </div>
    </div>
    <div>
      <p>Thanks for Watching!</p>
    </div>
  </body>
</html>'''
```

- Collect all children of a specific element:

Xpath = '/html/body/'*

- Collect all descendants of the body element:

Xpath = "/html/body//"*

- Collect all elements in the entire HTML document:

Xpath = "//"*

- Now we can specify relevant path by using *, /, //, [], @, searching for elements or attributes.
- Further we can use contain function for attributes containing multiple objects (e.g. <p class = 'class1 class2'>), to instead of direct match

XPATHS

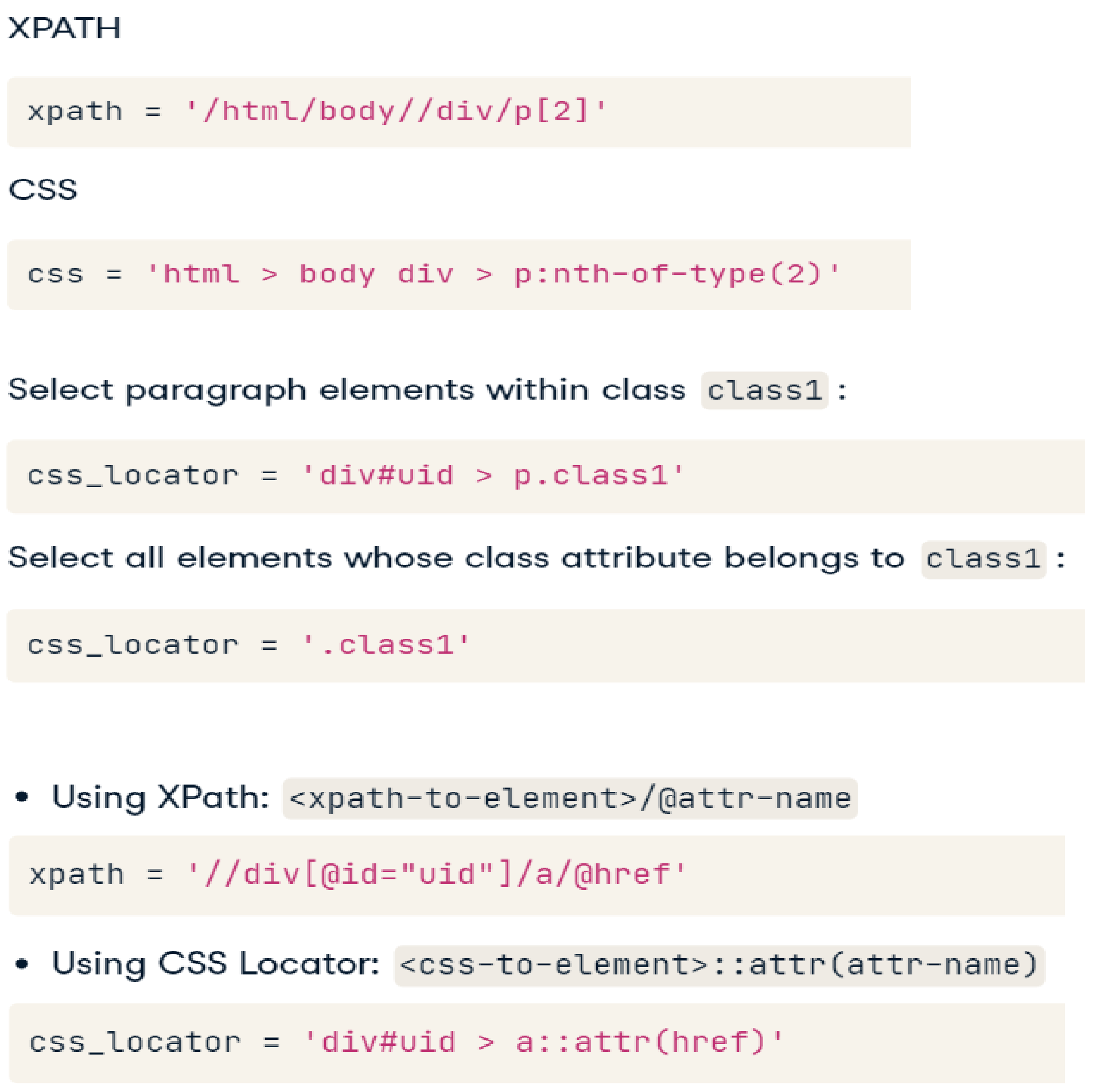
Example:

```
html = '''
<html>
 <body>
  <div id="div1" class="class-1">
   <p class="class-1 class-2">Hello World!</p>
   <div id="div2">
    <p id="p2" class="class-2">Choose
      <a href="http://datascience.com">Data Science!</a>!
    </p>
   </div>
  </div>
  <div id="div3" class="class-2">
   <p class="class-2">Thanks for Watching!</p>
  </div>
 </body>
</html>'"
```

- to select the paragraph element containing the phrase: "Thanks for Watching!", by using id attribute
- to select the paragraph element containing the phrase: "Hello World!", by using class attribute
- to select the href attribute value from the Data Science hyperlink (note that we want to get attribute content, not the element content)
- create an XPath which directs to all href attribute values of the hyperlink a elements whose class attributes contain the string "package-snippet"

CSS locator

- To find an element by class, use a period “.”
    - “p.class-1” selects all paragraph elements belonging to class-1
- To find an element by id, use a pound sign “#”
    - “div#uid” selects the div elements with id equals to uid



XPATH

```
xpath = '/html/body//div/p[2]'
```

CSS

```
css = 'html > body div > p:nth-of-type(2)'
```

Select paragraph elements within class `class1` :

```
css_locator = 'div#uid > p.class1'
```

Select all elements whose class attribute belongs to `class1` :

```
css_locator = '.class1'
```

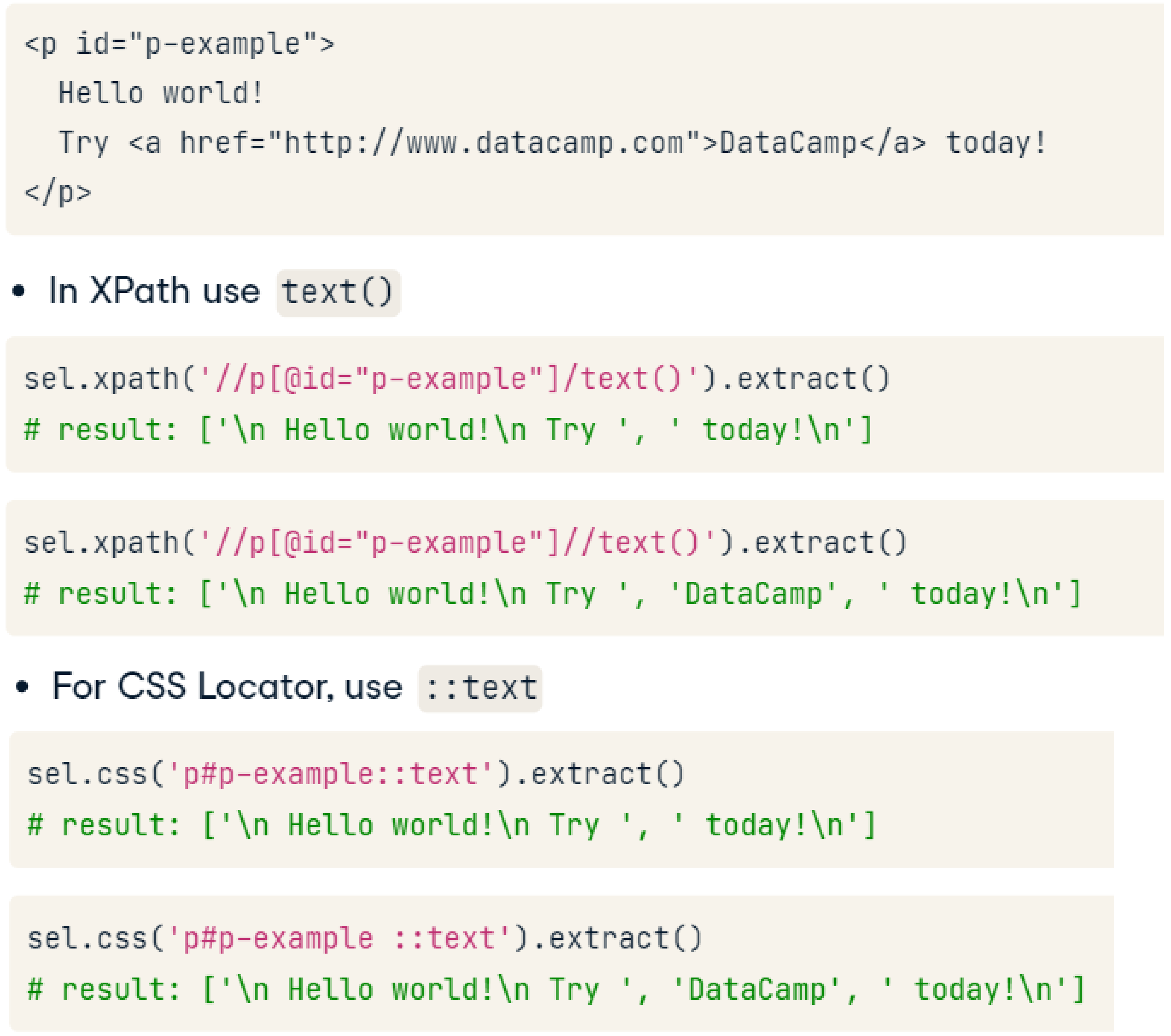
- Using XPath: `<xpath-to-element>/@attr-name`

```
xpath = '//div[@id="uid"]/a/@href'
```

- Using CSS Locator: `<css-to-element>::attr(attr-name)`

```
css_locator = 'div#uid > a::attr(href)'
```

Extracting texts Xpath vs CSS locator



```
<p id="p-example">
  Hello world!
  Try <a href="http://www.datacamp.com">DataCamp</a> today!
</p>
```

- In XPath use `text()`

```
sel.xpath('//p[@id="p-example"]/text()').extract()
# result: ['\n Hello world!\n Try ', ' today!\n']
```

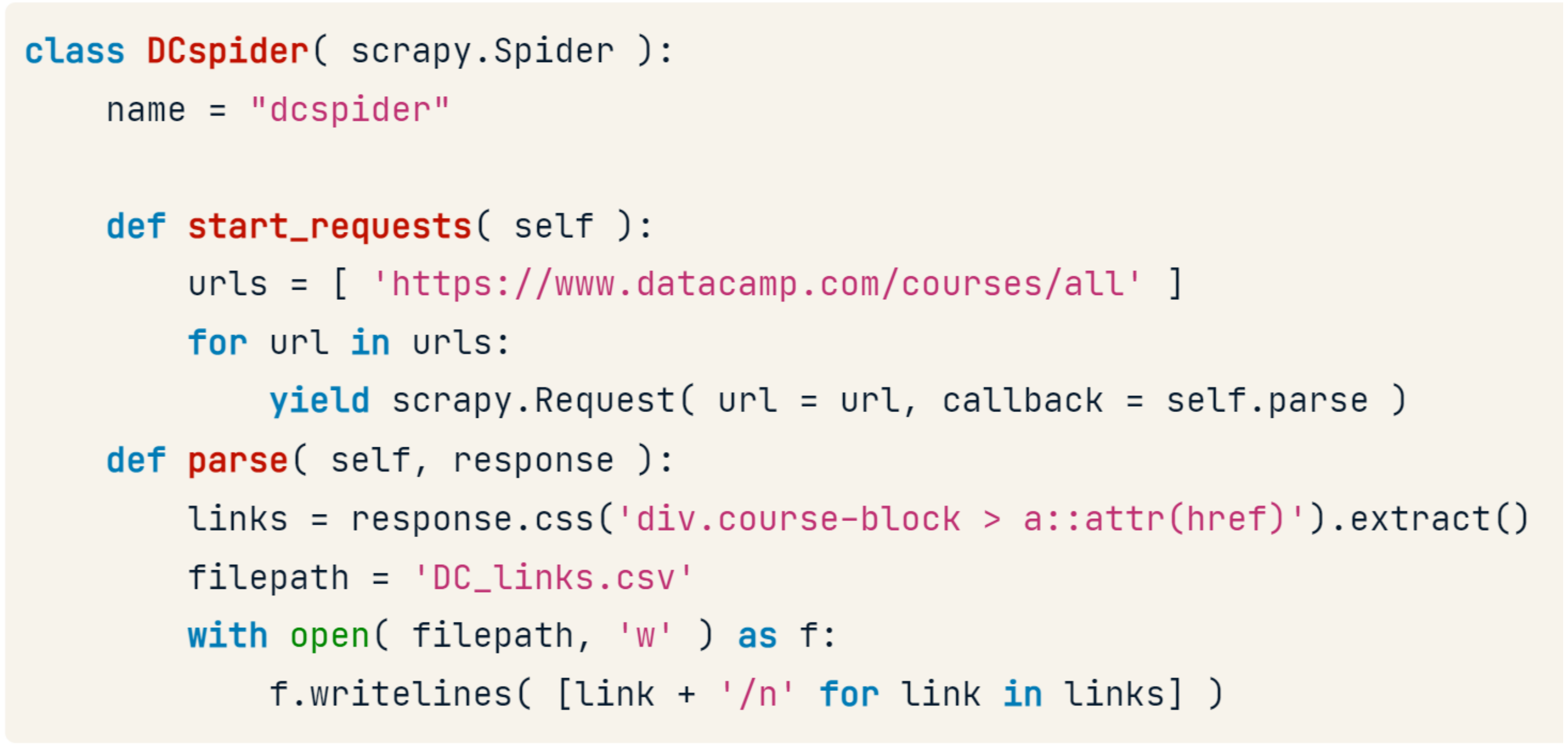
```
sel.xpath('//p[@id="p-example"]//text()').extract()
# result: ['\n Hello world!\n Try ', 'DataCamp', ' today!\n']
```

- For CSS Locator, use `::text`

```
sel.css('p#p-example::text').extract()
# result: ['\n Hello world!\n Try ', ' today!\n']
```

```
sel.css('p#p-example ::text').extract()
# result: ['\n Hello world!\n Try ', 'DataCamp', ' today!\n']
```

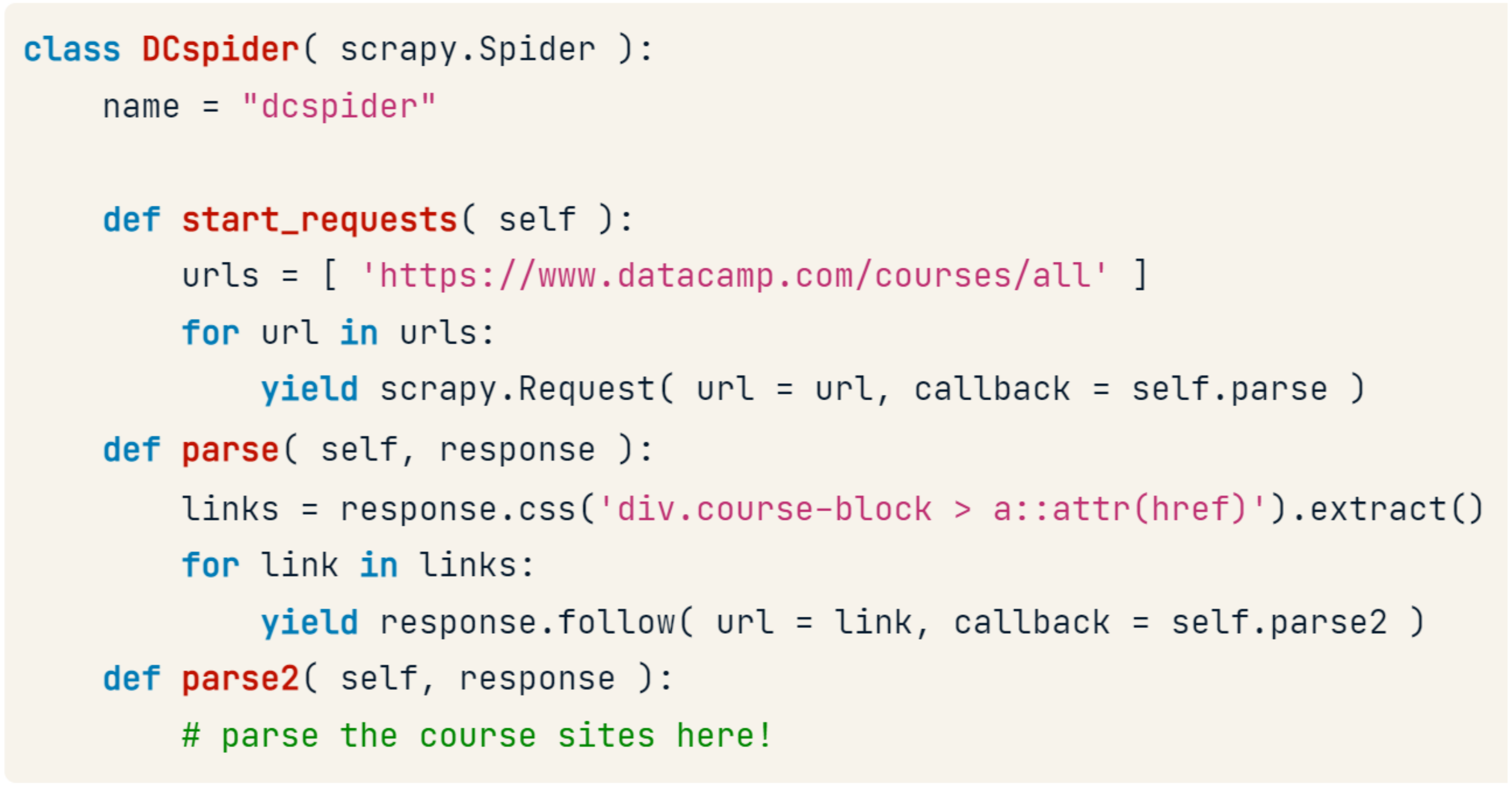
Scraping with Spider



```python
class DCspider( scrapy.Spider ):
    name = "dcspider"

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )
    def parse( self, response ):
        links = response.css('div.course-block > a::attr(href)').extract()
        filepath = 'DC_links.csv'
        with open( filepath, 'w' ) as f:
            f.writelines( [link + '/n' for link in links] )
```

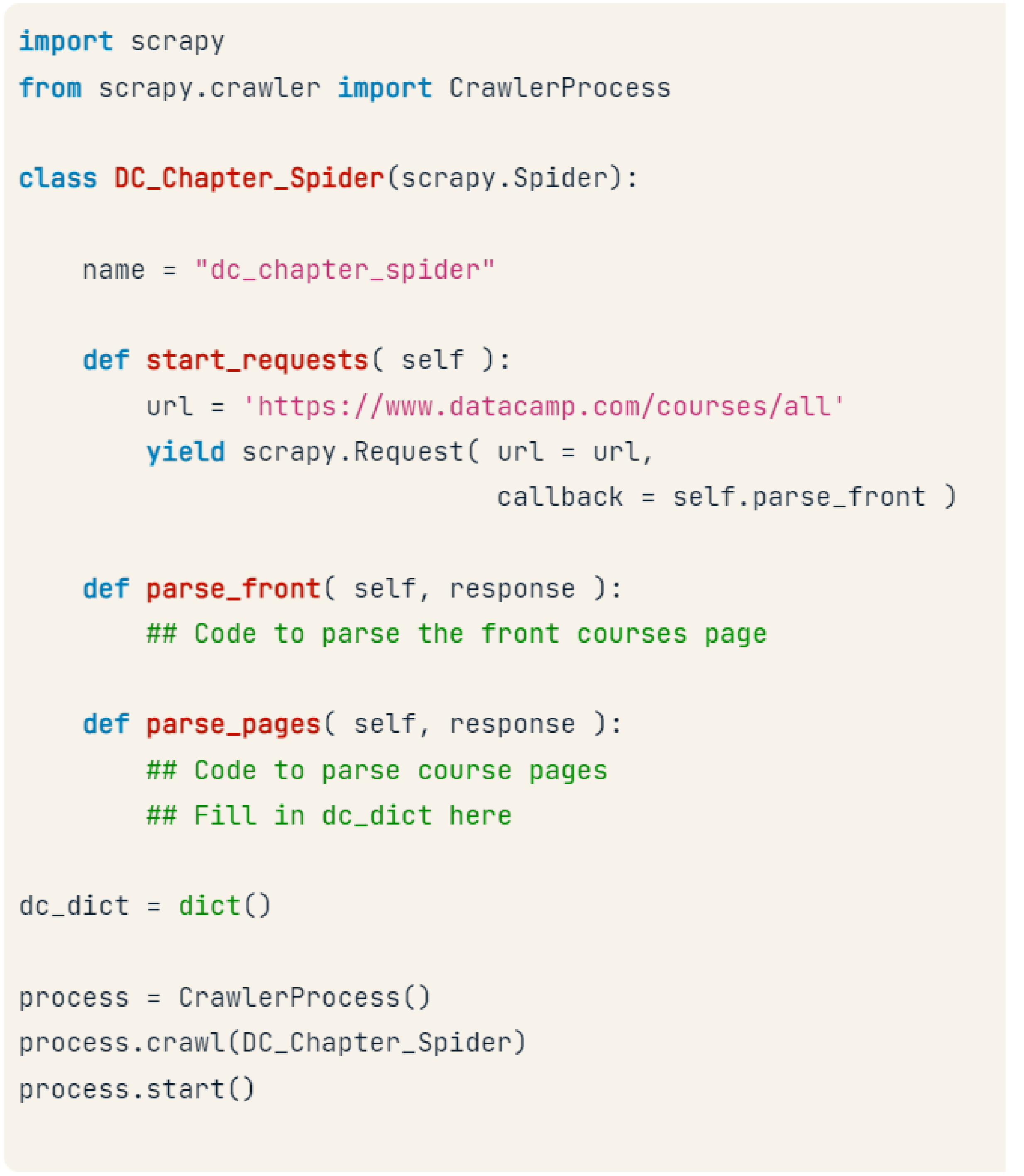
Scraping with Spider



```
class DCspider( scrapy.Spider ):
    name = "dcspider"

    def start_requests( self ):
        urls = [ 'https://www.datacamp.com/courses/all' ]
        for url in urls:
            yield scrapy.Request( url = url, callback = self.parse )
    def parse( self, response ):
        links = response.css('div.course-block > a::attr(href)').extract()
        for link in links:
            yield response.follow( url = link, callback = self.parse2 )
    def parse2( self, response ):
        # parse the course sites here!
```

Scraping with Spider



```python
import scrapy
from scrapy.crawler import CrawlerProcess

class DC_Chapter_Spider(scrapy.Spider):

    name = "dc_chapter_spider"

    def start_requests( self ):
        url = 'https://www.datacamp.com/courses/all'
        yield scrapy.Request( url = url,
                              callback = self.parse_front )

    def parse_front( self, response ):
        ## Code to parse the front courses page

    def parse_pages( self, response ):
        ## Code to parse course pages
        ## Fill in dc_dict here

dc_dict = dict()

process = CrawlerProcess()
process.crawl(DC_Chapter_Spider)
process.start()
```

Scraping with Spider

```python
def parse_front( self, response ):
    # Narrow in on the course blocks
    course_blocks = response.css( 'div.course-block' )
    # Direct to the course links
    course_links = course_blocks.xpath( './a/@href' )
    # Extract the links (as a list of strings)
    links_to_follow = course_links.extract()
    # Follow the links to the next parser
    for url in links_to_follow:
        yield response.follow( url = url,
                               callback = self.parse_pages )
```

```python
def parse_pages( self, response ):
    # Direct to the course title text
    crs_title = response.xpath('//h1[contains(@class,"title")]/text()')
    # Extract and clean the course title text
    crs_title_ext = crs_title.extract_first().strip()
    # Direct to the chapter titles text
    ch_titles = response.css( 'h4.chapter__title::text' )
    # Extract and clean the chapter titles text
    ch_titles_ext = [t.strip() for t in ch_titles.extract()]
    # Store this in our dictionary
    dc_dict[ crs_title_ext ] = ch_titles_ext
```

Practical session for Web scrapping: IMDB data