Web crawlers

A *web crawler* is a program that systematically browses the web. They are typically used by search engines to index pages and make the web easier to navigate. They can also be used to collect information. *Web scraping* is the process of extracting and storing information from the web in a strucured format that is easy to process.

A web crawler starts with a list of URLs to visit, known as *seeds*. A good seed page should link to many other pages with a topic relevant to your search. As the crawler visits these URLs, it identifies all relevant hyperlinks in each page and adds them to the list of URLs to visit, known as the *crawl frontier*.

Examples of web crawlers include those used by search engines like Google and Bing, known as Googlebot and Bingbot, respectively. Other examples include YaCy, a distributed search engine based on peer-to-peer networks and distributed under the GNU General Public License, as well as Apache Nutch, a Java-based open-source web crawler.

In general, web crawlers can be used for web scraping and data mining, web indexing (in the case of search engines), link validation, and web archiving. An example of a web archive is the Wayback Machine, a digital archive of the World Wide Web created by the Internet Archive, a nonprofit organization. Web crawlers can also be used for malicious purposes, such as email harvesting and penetration testing, which is the process of finding security vulnerabilities on a system.

Crawling policies

A *crawling policy* determines the behavior of a web crawler. Examples of crawling policies include selection policy, revisit policy, parallelization policy, and politeness policy.

The *selection policy* determines which pages the crawler should download. It is desireable for the crawler to download the fraction of pages which contain the most

relevant information to its search, given the enormous size of the web. This requires some kind of metric of importance for prioritizing which pages should be visited or downloaded first.

Examples of metrics for a page include number of backlinks, frequency of visits, the presence of certain URL keywords, as well as partial PageRank calculations. PageRank is the algorithm used by Google Search to rank websites in their search engine results. The basic idea behind PageRank is that more important websites are more likely to be linked to from other important websites.

The *revisit* policy determines when to check for changes to web pages. The web is not static, as pages and their content are constantly being created, updated, and deleted. Ideally, a web crawler should be able to estimate how often a page is updated to that it can revisit it at an optimal rate.

The *parallelization* policy determines how distributed web crawlers should be coordinated. This avoids duplication of work and repeated downloads of the same page when the same URL is found by multiple instances of a web crawler.

The *politeness policy* ensures that a web crawler's activity does not hamper the performance of a website for other users. Because web crawlers can send a large number of requests per second and download large amounts of data from a website, they can potentially overload its server, especially if multiple crawlers are crawling the same website.

Ideally, your web crawler should respect a website's robot inclusion and exclusion standards, identify itself when requested, respect the license terms of copyrighted material, and not interfere with the normal operation of the server. You can maintain a web page that discloses the robot's purpose in case of an inquiry.

The *robots.txt* is a robot exclusion standard that web crawlers and web robots read when visiting a website. It lists areas that should not be processed or scanned (hence excluded) in the web robot's search. It can also include a crawl delay that sets a maximum number of requests per second that can be made by a robot. More information is available at the robots.txt website.

Examples of robots.txt can be found for Google, Yahoo, and Facebook. The website for Columbia University, for example, as the following robots.txt file:

```
# ignore this line - 1
# for info on robots.txt syntax see
# http://www.searchtools.com/robots/robots-txt.html

User-agent: *
Disallow: /cgi-bin/
Disallow: /acis/whatsnew.html
Disallow: /httpd/reports/
Disallow: /itc/ccnmtl/assets/
```

The `User-agent: *` line signifies that this section applies to all robots. Each `Disallow:` line tells the robot that it should not visit a particular page or directory. The `cgi-bin` directory, for example, stores Perl and compiled script files.

You may notice that some robots.txt files link to a Sitemap or multiple Sitemaps. A *Sitemap* is a robot *inclusion* standard that consists of an XML file listing the URLs that are available for crawling. It can also provide additional information such as the last date a page was updated, how frequently a page is updated, how important certain pages are, and so on. This allows web crawlers to crawl the site more intelligently and efficiently. Take a look at the protocol for more information.

Extracting information

When a crawler is visiting a page, it needs to be able to automatically extract the information it is looking for. For this purpose we use something called *XPath*. XPath is a language for selecting nodes from an XML or HTML document. For example, suppose our XML document consists of the following:

```
<A>
  <B>
    <C></C>
  </B>
  <D></D>
```

```
</A>
```

A simple example of an XPath is `/A`, which selects the entire tree. Another simple example is `/A/B/C`, which selects C elements that are children of B elements that are children of A elements. The XPath `/A//C` would select all C elements that are descendants (children, grandchildren, and so on) of A elements.

Here are some useful examples of XPaths for selecting elements from an HTML document:

- `/html/head/title` selects the `<title>` element inside the `<head>` element of an HTML document.
- `/html/head/title/text()` extracts the text from the aforementioned `<title>` element.
- `//a` selects all `<a>` elements (hyperlinks) in the document
- `//div[@class="myclass"]` selects all `<div>` elements which contain an attribute `class` equal to the string `"myclass"`

Here is another example:

```
<?xml version="1.0" encoding="utf-8"?>
<wikimedia>
  <projects>
    <project name="Wikipedia" launch="2001-01-05">
          <edition language="English">en.wikipedia.org</edition>
          <edition language="German">de.wikipedia.org</edition>
          <edition language="French">fr.wikipedia.org</edition>
          <edition language="Polish">pl.wikipedia.org</edition>
    </project>
    <project name="Wiktionary" launch="2002-12-12">
          <edition language="English">en.wiktionary.org</edition>
          <edition language="French">fr.wiktionary.org</edition>
          <edition language="Vietnamese">vi.wiktionary.org</edition>
          <edition language="Turkish">tr.wiktionary.org</edition>
    </project>
  </projects>
</wikimedia>
```

The XPath `/wikimedia/projects/project/editions/*[2]` would select the following nodes:

```
<edition language="German">de.wikipedia.org</edition>
<edition language="French">fr.wiktionary.org</edition>
```

In other words, it selects the second child under the `editions` node of each `project` node under `/wikimedia/projects/` .

If you're using a web browser like Google Chrome or Mozilla Firefox, you can use the Chrome Developer Tools or Firebug HTML Panel, respectively, to extract the XPath of an element you point to on a page and its location in the document structure. To do this, right click on an element in a page and select `Inspect` or `Inspect Element` . Once a panel opens and displays the HTML structure of the page, right click on any element and select `Copy XPath` .

You can also use an alternative to XPath known as CSS Selectors. They work in a very similar way but instead select nodes based on their CSS style. Each stylesheet rule has a selector pattern that matches a set of HTML elements.

Now that you know the basics of how to extract data from a page, you can also learn how to extract the links for other pages you are interested, follow them and then extract the data you want for all of them.

Web crawling in Scrapy

Let's start writing our web crawler in Scrapy. Scrapy is an open-source web crawling framework written in Python and designed for web scraping. First, make sure you have Scrapy installed on your computer by entering `pip install scrapy` in your Terminal.

Now navigate to the directory where you would like to create your Scrapy project. Enter `scrapy startproject myproject` . This will create a project with the following directory structure:

```
myproject/
    scrapy.cfg
    myproject/
        __init__.py
        items.py
        pipelines.py
        settings.py
        spiders/
            __init__.py
```

The `scrapy.cfg` file is the deploy configuration file which contains configuration parameters and settings for your project. The directory where this file resides is the project root directory.

You can see that the project directory also contains a folder with the projects items, pipelines, and settings. It also contains a subdirectory called `spiders`, which is where you will store your spider programs or web crawlers. Navigate to the inner `myproject` directory by entering `cd myproject/myproject`.

Edit the project items file in your default Python code editor by entering `open items.py`. You should see the following code:

```
# Define here the models for your scraped items
#
# See documentation in:
# http://doc.scrapy.org/en/latest/topics/items.html

import scrapy


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

Items are containers that contain the data your web crawler will scrape in structured form. As you can see, they are declared by creating a class that inherits from `scrapy.Item` and defining its attributes as `scrapy.Field` objects.

In our example, we will be scraping comments from the Bwog website. For each comment, we want to store its content, author, date and time of posting, and number of upvotes and downvotes it has received. To do this, change your code to

```
import scrapy

class MyItem(scrapy.Item):
  author = scrapy.Field()
  up = scrapy.Field()
  down = scrapy.Field()
  datetime = scrapy.Field()
  content = scrapy.Field()
```

Now let's create the spider. Navigate to the directory that contains the project's spiders by entering `cd spiders`. Create a new spider by entering `touch myspider.py`, then open it in your default Python code editor by entering `open myspider.py`. Enter the following code:

```
import scrapy
from myproject.items import MyItem

class MySpider(scrapy.Spider):
  name = 'myspider'
  allowed_domains = ['bwog.com']
  start_urls = ['http://bwog.com']
```

This creates a class called `MySpider` that inherits from the Scrapy library's `Spider` base class. The `name` field is a unique identifier for our spider. The `allowed_domains` field should be set to a list of domains that you want your spider to stay in. If the spider comes across a link that leads to a domain outside this list, the spider will not follow it. Finally, the `start_urls` field defines the list of URLs where the spider will begin to crawl from, also known as the seed pages. Subsequent URLs will be generated from data contained in these pages.

The spider class should have a method called `parse` that will be called with the downloaded Response object of each start URL. The response is passed to the method

as the first and only argument. This method is responsible for parsing the response data and extracting scraped data (as scraped items), as well as determining which other URLs the spider should follow.

```
  def parse(self, response):
    for section in response.xpath('//div[@class="blog-section"]'):
      link = section.xpath('.//a/@href').extract_first()
      yield scrapy.Request(link, callback=self.parse_entry)
    next = response.xpath('//div[@class="comnt-btn"]//@href').extract_
first()
    yield scrapy.Request(next, callback=self.parse)
```

You can see that we have used a method of the response object called `xpath()`. `xpath()` returns a list of selectors, each of which represents the nodes selected by the XPath expression given as argument. `extract()` returns a unicode string with the selected data. `extract_first()` does the same thing but selects only the first element matching the XPath.

You can also see that we used the method `scrapy.Request()`. Scrapy creates `scrapy.Request` objects for each URL in the `start_urls` list and assigns them the `parse` method of the spider as their callback function. These Requests are scheduled, then executed, and scrapy.http.Response objects are returned and then fed back to the spider, through the `parse()` method. In our case, we are programatically selecting the link that leads from the front page of Bwog to older entries and telling our spider to follow it.

You may have also noticed that we called `scrapy.Request` on a method called `parse_entry`, which we haven't defined yet. To define this method, add the following code to our spider class:

```
  def parse_entry(self, response):
    for comment in response.xpath('//div[contains(@class, " comment-bo
dy")]'):
      item = MyItem()

      item['author'] = comment.xpath('./div[@class="comment-author vca
rd"]/cite/text()').extract_first()
```

```
        metadata = comment.xpath('./div[@class="comment-meta datetim
    e"]')
        item['up'] = int(metadata.xpath('./span[@data-voting-direction
    ="up"]/span/text()').extract_first())
        item['down'] = int(metadata.xpath('./span[@data-voting-direction
    ="down"]/span/text()').extract_first())
        item['datetime'] = metadata.xpath('./a/text()').extract_first().
    strip()

        paragraphs = comment.xpath('./div[contains(@class, "reg-comment-
    body")]/p/text()').extract()
        item['content'] = '\n'.join(paragraphs)

        yield item
```

This code selects all comments on a blog entry page and extracts the relevant information we wanted to store, including the comment's author, metadata, and content. It returns a Scrapy Item that contains these fields together with their values.

We are ready to see our web crawler in action. Navigate to the top directory of your project by entering `cd ../..`. Run the spider you created and store its output in a `comments.json` file by entering `scrapy crawl myspider -o comments.json`. You should see your terminal being filled with output as the crawler extracts comments from Bwog for older and older entries. You can stop the crawling process at any time by entering `Ctrl + C`.

To view the comments that the spider stored in the crawling process, enter `open comments.json` in the same project root direcotry. You should see that the crawler stored the Bwog comments in a structured JSON format.

Next steps

There are many other features you can take advantage of when using Scrapy. Scrapy includes an interactive shell console for trying out CSS and XPath expressions to scrape data, for example. This is very useful for writing and debugging your spiders on different web pages.

Scrapy also supports exports to multiple formats, including JSON, CSV, and XML, as well as multiple backends, including FTP and S3. You can store the scraped items in a database like MongoDB as well.

After an item has been scraped by a spider, it is sent to the Item Pipeline which processes it through several components that are executed sequentially. Each item pipeline component receives an item and performs an action over it, also deciding if the item should be dropped and no longer processed.

Typical uses of item pipelines include cleansing HTML data, validating the scraped data (e.g. by checking that items contain certain fields), and checking for duplicate items before they are processed further down the pipeline.

To learn more about Scrapy and the web scraping process, you can visit the official Scrapy tutorials and documentation online. You can also find the code for this tutorial on Github.