Preparing Input Data for EFDC Simulations with Scrapy

Scrapy is a python framework for crawling websites and extract structured data. In this report, we show how to use scrapy to collect data from National Oceanic and Atmospheric Administration (NOAA) and U.S. Geological Survey (USGS).

In Scrapy, we are particularly interested in two python classes:

Spiders are classes which define how a certain site (or group of sites) will be scraped, including:

- Which links to crawl
- How to extract structured data from the pages

Items are the containers used to store the structured data extracted from the Spiders.

Pipeline to scape data from websites

The basic steps for gathering data from websites are:

1. Construct the URLs
2. Get responses from URLs
3. Parse responses to extract data
4. Process the data

In scrapy, spiders are subclassed from `scrapy.spider.BaseSpider`, with these class properties:

- `name`: the unique name that identifies the spider
- `start_urls`: a list of URLs where the spider will begin to crawl from.
- `parse()`: the method which will be called by the `Response` object of each start URL.

Items are subclassed from `scrapy.item.Item`. They are used in Spiders to store extracted data. Items are declared in a straightforward way with specific fields based on the data, for example:

```
from scrapy.item import Item, Field

class Station(Item):
```

```
    id = Field()
    name = Field()
```

The `Station` contains two fields: `id` and `name`.

A simple spider

Here we show a simple example using scrapy. Our goal is to download the September data from the NDBC website:

> http://www.ndbc.noaa.gov/data/stdmet/Sep/dpia1.txt

- First, we need to create a new Scrapy project

Enter a directory and run:

```
scrapy startproject noaa_scrapy
```

This creates the basic structure in a folder called “noaa_scrapy”.

- Second, implement a spider

Before it will do anything, we need to define the spider.

We create the spider named “SimpleSpider”:

```
from scrapy.spider import BaseSpider
from scrapy.http import Response

class SimpleSpider(BaseSpider):
    name = "SimpleSpider"
    start_urls = [
        "http://www.ndbc.noaa.gov/data/stdmet/Sep/dpia1.txt"
    ]

    def parse(self, response):
        filename = response.url.split("/")[-1]
        open(filename, 'wb').write(response.body)
```

The `SimpleSpider` subclassed from `BaseSpider`, with the unique name “SimpleSpider”, `start_urls` are the URLs we are going to crawl. `parse()` receive the response and save the `response.body` to the file.

- Test the spider

To run our spider, go to the project's top level directory and run:

```
scrapy crawl SimpleSpider
```

Under the hood, scrapy creates a `scrapy.http.Request` object for the URL in the `start_urls` of the `SimpleSpider` (which is "http://www.ndbc.noaa.gov/data/realtime2/DPIA1.txt"), then the request is scheduled and executed, the `scrapy.http.Response` object is returned through the `parse()` function. In `SimpleSpider`, the `parse()` function will write the response body to the file DPIA1.txt. The file looks like:

```
#YY  MM DD hh mm WDIR WSPD GST  WVHT   DPD   APD MWD   PRES  ATMP  WTMP  DEWP  VI
S  TIDE
#yr  mo dy hr mn degT m/s  m/s     m   sec   sec deg    hPa  degC  degC  degC  nm
i    ft
2013 09 01 00 00 267  1.3  2.0 99.00 99.00 99.00 999 1011.9  29.3 999.0  23.2 99.
0 99.00
2013 09 01 01 00 248  1.5  2.1 99.00 99.00 99.00 999 1012.0  28.7 999.0  23.9 99.
0 99.00
2013 09 01 02 00 260  2.4  3.6 99.00 99.00 99.00 999 1012.3  28.5 999.0  24.2 99.
0 99.00
....
```

The Item class

In the previous version of `SimpleSpider`, `parse()` is quite trivial. There is no need to define a `Item` class. However, the `Item` class allows structurize the raw data and is convenient for further processing.

Define an item

We can then define the Item class `Record` as below according to the content:

```
from scrapy.item import Item, Field

class Record(Item):
    sid = Field()
    time = Field()
    wdir = Field()
    wspd = Field()
    gst = Field()
    wvht = Field()
    dpd = Field()
```

```
    apd = Field()
    mwd = Field()
    pres = Field()
    atmp = Field()
    wtmp = Field()
    dewp = Field()
    vis = Field()
    tide = Field()
```

Extract in parse()

With the definition of `Record`, we can add the following lines in the `parse()`:

```
def parse(self, response):
    recordlist = []
    for datum in response.body:
        split_list = datum.split()
        [year, month, day, hour, min,
         wdir, wspd,gst, wvht, dpd,
         apd, mwd, pres, atmp, wtmp, dewp, vis, tide] = split_list

        item = Record()
        item['sid'] = self.stationID.lower()
        item['time'] = time
        item['wdir'] = wdir
        item['wspd'] = wspd
        item['gst'] = gst
        item['wvht'] = wvht
        item['dpd'] = dpd
        item['apd'] = apd
        item['mwd'] = mwd
        item['pres'] = pres
        item['atmp'] = atmp
        item['wtmp'] = wtmp
        item['dewp'] = dewp
        item['vis'] = vis
        item['tide'] = tide
        recordlist.append(item)
```

In the funciton `parse()`, each line will be turned into a `Record` object and stored in the `recordlist` for further processing. For example, we can serialize them into a database.

A close look at the single URL case

In practice, we need to collect data from different data sources on the NDBC website. To achieve this, we need to take a close look at how the data is stored on the NDBC sites.

There are two types of data for a station in the NDBC database: the recent data and the historical data.

1. Recent data:
    - Data for last 24 hours The URL is:

    > http://www.ndbc.noaa.gov/data/hourly2/hour_nn.txt

    where *nn* ranges from 00 to 23. The file stores the data for every data in the given hour, for example:

    - Data for last 5 days

    > http://www.ndbc.noaa.gov/data/5day2/sid_5day.txt

    where the *sid* is the station number.

    - Data for last 45 days

    > http://www.ndbc.noaa.gov/data/realtime2/sid.txt

    where the *sid* is the station number.

2. Historical data:

Data older than 45 days are regarded as historical data at NDBC sites.

> http://www.ndbc.noaa.gov/view_text_file.php?
> filename=idstring.txt.gz&dir=data/historical/swdir/

where the *idstring* is to identify the station and year. For example, “DPIA1d2005” means the whole data from station DPIA1 in year 2005.

Selector: extract data based on XPath expressions

Given a station, we need to verify if the historical data exist in a given year. Fortunately, we can get the information from the website.

Let’s take the station DPIA1 for example. Following the link

> http://www.ndbc.noaa.gov/station_history.php?station=DPIA1

We can see that there is a list of all the available historical data for station DPIA1. Our task is to extract the data from the web page. Scrapy provides a XPath-style mechanism called selectors to extract the data.

Here are some examples of XPath expressions and their meanings:

- /html/head/title: selects the < title > element, inside the <head > element of a HTML document
- /html/head/title/text(): selects the text inside the aforementioned < title > element.
- //td: selects all the < td > elements
- //div[@class=”mine”]: selects all div elements which contain an attribute class=”mine”

With the help of XPath inspection tool, we can easily find our target. Here is the code to extract a list of years with data from the link above:

```
       prev_year_list = [int(yr)
       for yr in hxs.select('/html/body/table[2]/tr/td[3]/ul/li[2]/ul/li[1]/a/tex
t()').extract()
            ]
```

Generate the URLs

In general, there are different patterns in URLs. A single URL won’t handle all cases. One practical question is how to generate the URLs for several types of data. In Scrapy, we need to write a function for this. The function that generates the URLs is stored in the list `start_urls`.

Yield the next request with different parsing method

We also note that the format of *24-hour recent time data* is different from other data. As a result, we have to prepare two `parse()`: `parse_hourly()` for *24-hour real time data* and `parse_normal()` for others.

The question is how scrapy knows which parse function the `Request` should be sent to. A callback function can provide the required information. We can determine the target parse function according to the url. The parsing is delegated to `parse_hourly()` and `parse_normal()` via the callback function. The `parse()` is only the trigger for the next step.

After the `parse()` of a request, we can yield the next request for next parsing.

So, we add related functions and extend the `parse()` in our spider:

```
class SimpleSpider(BaseSpider):

    ...

    def generate_next_url(self):
        # Generate next url, save in self.next_url

    def dispatch(self):
        if "/hourly2/" in self.next_url:
            return Request(next_url, callback=self.parse_hourly)
        else:
            return Request(next_url, callback=self.parse_normal_day)

    def parse(self, response):
        self.generate_next_url()
        return self.dispatch()

    def parse_hourly(self, response):
        #parsing the hourly data
        self.generate_next_url()
        return self.dispatch()

    def parse_normal(self, response):
        #parsing normal data
        self.generate_next_url()
        return self.dispatch()
```

Using scrapy in other programs

The scrapy can be called from other program. We need to add a constructor for `SimpleSpider` to accept the parameters.

The code is shown below:

```
def __init__(self,
             stationID="DPIA1",
             sDate="2012-10-05 00:00:00",
             eDate="2012-10-05 23:00:00",
             **kwargs):
    super(Spider, self).__init__(self.name, **kwargs)
    self.stationID = stationID
    self.startDate = parser.parse(sDate)
    self.endDate = parser.parse(eDate)
    ...
```

To run scrapy from the command line by:

```
scrapy crawl -a  stationID=sid  -a startTime=stime -a endTime=etime SimpleSpider
```

The parameters *{'stationID'=sid, 'startTime'=stime, 'endTime'=etime}* will be passed to the Spider constructor. The *sid* is the station id, *stime* is the start time, *etime* is the end time.

Scripts to prepare EFDC input data

The Environmental Fluid Dynamics Code (EFDC Hydro) is a state-of-the-art hydrodynamic model that can be used to simulate aquatic systems in one, two, and three dimensions. A toolkit that helps to automate the process to download and preprocess the input data for EFDC is of great value to the EFDC modelers. Here we show how to prepare the input data for a particular EFDC modeling effort.

Data sources

In this particular case, EFDC requires input files for 3 forcing conditions: river discharge, wind, and open boundary water level. For Mobile Bay,

1. River discharge: use the sum of daily discharge values at the following 2 locations.

USGS 02469761 Tombigbee River at Coffeeville L&D near Coffeeville, AL (http://waterdata.usgs.gov/nwis/dv?referred_module=sw&site_no=02469761) USGS 02428400 Alabama River at Claiborne L&D near Monroeville (http://waterdata.usgs.gov/nwis/dv?referred_module=sw&site_no=02428400)

1. Wind: use the hourly wind data at NDBC station DPIA1

NDBC station DPIA1 (http://www.ndbc.noaa.gov/station_history.php?station=dpia1)

1. Water level at open boundary: we usually estimate water level OBCs using the observed water level at the NOAA station 8735180

NOAA station 8735180 (http://tidesandcurrents.noaa.gov/data_menu.shtml?stn=8735180%20Dauphin%20Island,%20AL&type=Historic+Tide+Data).

Scripts based on Scrapy

There are three spiders to grab three types of data respectively.

riverspider.py defines the `RiverSpider` to compute the sum of river discharges. Given a date interval, `RiverSpider` grabs the data from 02469761 and 02428400 from USGS site then write out the sum to the output file *qser.inp*.

windspider.py defines the `WindSpider` to grab data from station DPIA1. Given a date interval, `WindSpider` grabs data from NDBC then write out to the output file *wser.inp*.

waterspider.py defines the `WaterSpider` to grab data from station 8735180. Given a date interval, `WaterSpider` grabs data from NOAA then write out to the output file *pser.inp*.

There are three scripts to run the three spiders respectively. For example, you can launch the `RiverSpider` to grab river data between the date 2010-05-01 and 2010-05-07 by running:

```
$python run_river.py “2013-05-01 00:00:00” “2013-05-07 00:00:00”
```

Similarly, `run_wind.py` and `run_water.py` are to launch the `WindSpider` and `WaterSpider` respectively.

```
$python run_wind.py “2013-05-01 00:00:00” “2013-05-07 00:00:00”

$python run_water.py “2013-05-01 00:00:00” “2013-05-07 00:00:00”
```

The script `run_all.py` launches the three spiders one by one with provided date range.

```
$python run_all.py “2013-05-01 00:00:00” “2013-05-07 00:00:00”
```

To build your own workflow using these scripts, you can simiply specify the start and end date time range as the parameters to `run_all.py`. The date string is in the format of

yyyy-mm-dd HH:MM:SS