Using Scrapy to acquire online data and export to multiple output files

Matthew J. Holland[*]

Abstract

In this document the task of acquiring large volumes of data (on the order of many millions of entries) encoded into web page HTML source code and URLs is discussed using the Python-driven Scrapy framework. Using a generalized problem requiring "scraping" and processing online tabular data, and subsequently outputting it into some desired form spanning multiple files, each created in a sequential manner when the previous file has reached some pre-defined capacity, a practical look is offered of the types of problems Scrapy can be used to deal with.

Introduction

Here we describe a full working prototype of a fairly simple web-scraping program developed using the Scrapy framework (written in Python). The program was designed with two main functions: (1) to acquire a great deal of archival public data presently available only in the form of web page HTML source code, and (2) to acquire new data as it is added to the online repository. My research involves statistical learning and naturally the difference between the two functions parallels the difference between facing a "batch" problem and a "sequential learning" problem. That said, having succeeded in getting the first function to work, the second can be realized with an almost trivial extension of code, so this document will give an overview of a program which scrapes and processes the past archive data only.

The specific content being downloaded of course pertains to a specific research problem, and while interesting (to the author) getting bogged down in such problem-specific details can be thought to detract from the lucidity of this document. As such, the "content" being handled will be referred to using totally general terms more related to the data structure than the specific contents of that structure.

While this document does give a reasonably detailed exposition of what elements to include in a Scrapy program and why, Scrapy is quite a rich environment to work in and thus every detail of every object assuredly cannot be treated. The work below was done using Scrapy 0.16.3 in Python 2.7 (Scrapy requires 2.x), and a well-written body of documentation exists online for current and past versions [2]. For users with no experience at all with Scrapy, at a bare minimum the official tutorial is a must-read [3]. A web search for tutorials will yield several similar results, all quite basic but with unique elements that offer excellent practice for simple scraping programs. As well, getting Scrapy installed can actually be a monstrous hassle,

[*]Affiliation: Mathematical Informatics Lab, Graduate School of Information Science, Nara Institute of Science and Technology, Nara, Japan. Contact: `matthew-h ATMK is.naist.jp`

so the author has put together a very non-formal document detailing what is necessary to get it set up on a 64-bit Windows machine [1].

Program objectives

Let's discuss in more detail precisely what we want this program to do for us. We consider the available inputs and our desired outputs.

Available inputs

- Data stored exclusively in a tabular structure, where rows denote unique datum.
- One table per URL, with all data encoded into the web page HTML.
- The data comes in varying time scales (10min/hourly/daily observations); each datum represents one observation at a given time scale, while each URL contains either one day's worth of observations at that time scale (in the 10min or hourly cases) or a month's worth of data for lower time resolutions.
- The data is also made at numerous observation sites, which are characterized by a "region ID" and a unique "site ID."
- ID data as well as observations dates are encoded into the URL itself.
- Each site has anywhere from several hundred thousand to upwards of 1.5 million rows of data for each time scale.
- There are several hundred sites from which data can be collected.

Desired outputs

- A recreation of the tabular data structure in a format more tractable from a statistical analysis standpoint (.csv, .txt, etc.).
- In addition to the HTML-encoded tabular data, for each output entry we would like to add the information encoded in the URL, in the form of a standardized date stamp as well site and region ID numbers.
- While a detailed folder structure is not necessary, file nomenclature is critical; considering the volume of data points, for each site and each timescale, one output file should be generated for each year's worth of data, and the file naming should reflect region, site, timescale, and year.
- Again due to the volume of data, as a bare minimum, given a certain timescale the program should generate all the required output for a single site over the entire span of interest.
- Ideally, since we may not need data from every site, given a timescale we would like to give our program a list of sites to visit, and have it automatically do the main routine for each site in order, generating the above output.

Thanks to the flexible nature of the Scrapy environment, generating a program which successfully generates all of our desired outputs is an entirely doable task. Using Scrapy requires only a basic knowledge of XPath selectors and Python, and our discussion henceforth assumes that the reader looking to emulate the results (in their own problem contexts, of course) both has Scrapy installed and possesses knowledge of the fundamentals of XPath selectors and Python. That said, the author is a researcher in the field of mathematical information engineering, and is not a programmer, so a large portion of readers are probably capable of building far more efficient software, and are wholeheartedly encouraged to do so. Consider this document an introduction to what sorts of problems Scrapy-built programs can be adapted to solve.

A full run-through

Here we start from the very beginning of our project to provide a start-to-finish run-through of a working scraping program (done on 64-bit Windows). From the command prompt we navigate to `../python27/scrapy`, the latter being a folder we created, and start a new project we shall call "table scrape."

```
scrapy startproject table_scrape
```

From the folder `/table_scrape/table_scrape` (note the double-stack) we now can begin working away at our code. The code differs only slightly depending on which timescale we use, and for concreteness we will consider the case of hourly data.

Items

So long as we know what output we want, `items.py` is by far the easiest thing to write. Let us say that the online tabular data has five columns, one being "time," and the remainder being data which we will say is of generic types A, B, C, and D. In addition we want region and site data which we will encode into a single "cell." As we take all the data we scan and pile it into a single output file at a time, a single item `hourlyItems` is sufficient:

```
from scrapy.item import Item, Field

class hourlyItems(Item):
    date = Field()
    time = Field()
    reg_site = Field()
    data_a = Field()
    data_b = Field()
    data_c = Field()
    data_d = Field()
    pass
```

Having this basic structure in place, we're already prepared to build our spider. This is where the *vast* majority of the work must be done, but in reality we go back and forth between the spider and the item pipeline which controls the actual output our program generates. Thus we have merged them into one *long* section to cover the program-building process in the most intuitive order possible.

The spider and item pipeline

We limit our program to a single spider. The process is as follows: *open* spider, *read* a $(site, region)$ pair, *read* a [date*i*, date*j*] date range, *scrape* all dates for the given site/region pair, then *repeat* for all such pairs until have reached the end of the list, then *close* the spider.

That is, we have a single spider which only technically runs once, but has recursive functionality which lets us do all the work that needs to be done. Over that recursion, we want it to "hop" from observation site to observation site, and from date to date in a systematic, perfectly sequential manner. By hand the process would be unthinkably tedious and easily prone to error in the copy-paste operation, thus a perfect job for a machine.

Initial spider work

Python's built-in CSV functions are excellent, so we will make use of them here when even a simple text file would do. We need to give our spider a list of observation sites to hop to, noting that sites are characterized by a two-digit region code and a four-digit site code. Let's call the file `orders_up.csv`, place it in our `../table_scrape/spiders` folder, and have it list out in a single column our six identifying digits of interest:

```
23-2069
23-2929
25-6684
...
```

We will thus be using the `csv` module, and later will make *extensive* use of the `Request` object, so our spider `table_scrape_spider.py` will begin with:

```
from scrapy.spider import BaseSpider
from table_scrape.items import hourlyItems
from scrapy.selector import HtmlXPathSelector
from scrapy.http import Request
import csv
```

We only need one spider, which means we only need one spider class. The class we shall call `hourlySpider`, and the spider will be named uncreatively `table_scrape_spider`. Let's say the domain name of the site we plan to use is `datsianta.com`. We begin with

```
class hourlySpider(BaseSpider):
    name = "table_scrape_spider"
    allowed_domains = ["datsianta.com"]
```

By far, the trickiest part of designing the spider is getting the URL-hopping mechanism to work flawlessly, since the URL contains many different elements, there is a lot of playing around with strings. We will see many opportunities to incorporate C code into our program, but here everything will be written in Python. The URL has the following structure (generalized for this example):

```
http://www.datsianta.com/timescale_h.php?region=XX&site=XXXX&y=XXXX&m=XX&d=XX

TXT_1 REGION(XX) TXT_2 SITE(XXXX) TXT_3 YEAR(XXXX) TXT_4 MONTH(XX) TXT_5 DAY(XX)
```

where `TXT_i` refers to what we will henceforth refer to as the *i*th "block," often expressed just using `b`. Since we are assuming a given time scale, we do not treat the timescale element of the URL as a variable.

To make things explicitly clear (as well as flexible) we will incorporate start and end dates for a given site's scan into the spider (and later the item pipeline) code. As well, we populate a list of our region/site pairs to visit from the "orders" CSV we wrote. It is important to realize that while `Requests` will be parsed over and over again, our spider will only be "opened" and "closed" once, meaning that it will only be initialized once. We thus initialize things we will use in the future right here:

```
    def __init__(self):
        # Pre-run settings (Spider)
        self.start_yr = "1988" # Str, 4 chars. Spider AND pipeline.
        self.start_mo = "01"   # Str, 2 chars. Spider ONLY.
        self.start_da = "01"   # Str, 2 chars. Spider ONLY.
        self.lim_yr = 2012     # Int. Spider AND pipeline.
        self.lim_mo = 12        # Int. Spider ONLY.

        self.days_index = days_dict() # dictionary for end-of-month check.

        self.orderlist = []
        with open("orders_up.csv",mode="r") as csvfile:
            orders = csv.reader(csvfile)
            for i in orders:
                self.orderlist.append(i[0])
```

In the recursion section, we will have a condition which uses a function to check whether, given the year, month, and day, that particular day is the last day of the month or not. In doing that, to remove clutter we initialize a simple dictionary, `days_index`. The function `days_dict()` could be a method or could be written as a global function, but in any case we'll write it as

```
def days_dict():
    mydict = {}
    for i in [1,3,5,7,8,10,12]:
        mydict[i] = 31
    for i in [4,6,9,11]:
        mydict[i] = 30
    return mydict
```

noting that February is not included. The reason for that is because we will inevitably need to deal with leap years, and to do that we will later employ an additional function. Now we are ready to make our initial request method. We will name our parsing function `hourly_parse()` and thus our spider's initial request will be directed using the method `start_requests`,

```
region=XX&site=XXXX&y=XXXX&m=XX&d=XX
    def start_requests(self):
        ini = Request("http://www.datsianta.com/timescale_h.php?region="+
                      (self.orderlist[0])[0:2]+"&site="+(self.orderlist[0])[3:]+
                      "&y="+self.start_yr+"&m="+self.start_mo+"&d="+self.start_da,
                      callback=self.hourly_parse)
        return [ini]
```

where we note that we are simply plucking the region and side IDs from the string extracted from the CSV file.

Parse function, prep work

Let's show the basic structure of the `hourly_parse` method, which is where the bulk of our content will go.

```
    def hourly_parse(self, response):

        # SEC 1: prep work.

        ### full node-set of XPath Selectors

        ### URL of scraped page to be broken down
        ### URL fragments
        ### dictionary for end-of-month check.

        # SEC 2: recursive scraping.

        ### a series of IF/ELSE conditions

        # SEC 3: main data-scraping routine
```

Using this as a guide, we will fill in the various sections as we go along, as well as define the additional functions needed to realize the functionality we need, though in the order of SEC 1, SEC 3, and finally SEC 2. Thus we begin with the "prep work" section. First the full node-set of XPath selectors and the URL fragments to be used (assuming the URL structure given above):

```
        hxs = HtmlXPathSelector(response) # full node-set of XPath Selectors

        url_f = str(response.url) # URL of scraped page to be broken down
        url_b1 = url_f[0:48]      # URL fragments
        url_region = url_f[48:50]
        url_b2 = url_f[50:56]
        url_site = url_f[56:60]
        url_b3 = url_f[60:63]
        url_year = int(url_f[63:67])
        url_b4 = url_f[67:70]
        url_month = int(url_f[70:72])
        url_b5 = url_f[72:75]
        url_day = int(url_f[75:77])
```

paying attention to the fact that since we have already gone ahead and turned the date variables into integers. We will need to convert them back to two-digit strings (even for months less than 10) later. That's it for the first section of our parse function. We will skip the second section *for now* and come back to it later. First we want to write the main routine to be performed by our spider, which is the data scrape operation itself.

Parse function, main scrape routine

Naturally, we want to get the data which is encoded into an HTML table within the page's source code. Using an application like Firebug (or a great deal of patience for reading over markup tags), we can identify the data structure inherent to the pages we want to scrape. For our example, let's say all our data rows are in a `table` element containing the attribute `class` with value `big_table`. Each row is delimited with `tr` tags, and cells with `td` tags.

We are *not* creating new CSVs for every page, and thus there is no need to read headers. Let's say that just the first row has headers, and the rest are rows of data. We then use appropriate

XPath expressions (using "nested selectors") to generate the data we want to work with in a clear way:

```
        # SEC 3: main data-scraping routine
        table = hxs.select("//table[@class='big_table']")
        data_rows = table.select("tr[position()>1]")
```

The code to follow `data_rows` is fairly standard across most Scrapy applications of this sort. Given a single URL to scrape, using a `for` loop we scan over each row and make one whole "entry." The entry has several data types, and these data types are organized according to the `Fields` we defined in `items.py` using our item `hourlyItems`. It should be noted that `hourlyItems` is a dictionary, as we will see below. For each row of data scraped (and thus each row of data output), we overwrite the previous dictionary assigned arbitrarily to `store`, populate it with all the data we scrape, append that dictionary into our "list of rows" called `items`, and then repeat the process until we're out of rows on the page our URL brought us to.

While in the case of a single URL a very simply function is fine, since we will be going over many URLs, we need for this to `hourly_parse` function to be a *generator* function, which is an "iterable object" in Python that uses `yield` to essentially "return" a set of values that Scrapy can just look at once, process (and potentially output as a CSV), and then discard from memory (good considering the volume of data we have). Thus, the final code for this section will come out as follows:

```
        # SEC 3: main data-scraping routine
        table = hxs.select("//table[@class='big_table']")
        headers = table.select("tr[position()<=2]")
        data_rows = table.select("tr[position()>2]")
        items = []
        for i in data_rows:
            store = hourlyItems()
            store["date"] = str(response.url[63:67]+"/"+response.url[70:72]+
                                "/"+response.url[75:77])
            store["time"] = slago(str(i.select("td[1]/text()").extract()))
            store["data_a"] = slago(str(i.select("td[2]/text()").extract()))
            store["data_b"] = slago(str(i.select("td[3]/text()").extract()))
            store["data_c"] = slago(str(i.select("td[4]/text()").extract()))
            store["data_d"] = str(trans(str(i.select("td[5]/text()").extract())))
            store["reg_site"] = url_region+"_"+url_site
            items.append(store)

        for item in items:
            yield item
```

Here, `slago` is a simple function

```
def slago(n):
    if n == "[u'----']":
        return "9999"
    else:
        return (str(n))[3:-2]
```

which clearly takes "null" data entries which are denoted on the website by a series of four hyphens, and turns them into the string 9999. This is not strictly necessary, but assuming 9999 is an impossible value for those data types, it can be handy to have the output in uniformly numerical form for later analysis. When Scrapy uses the XPath expression `text()` with a nested `extract()`, it always gives the `output` in the form `[u'output']`. The function `trans` is ready to translate a single Japanese character (of course not realistic, but pedagogically illustrative) should it be the case that such character data was scraped, whose Unicode code point Scrapy kindly provides by default for us. The function returns the number `1` should it be found:

```
def trans(n):
    if n == "[u'\u7a4f']":
        return 1
    else:
        return 9999
```

The fact is, the type/format of the data which one encounters is incredibly problem-specific, and the above are just a few examples which hopefully reflect the flexibility we have in dealing with such variety. With that, we have all we need for our main routine. Note that we still have *no output*, and *no URL-hopping capacity*. We will tackle the latter first.

Parse function, recursive element

We now return to the second section of our `hourly_parse` function. As this is a generator function, we do not `return` anything, but rather we `yield` it. The key to the recursion here is that we yield a `Request` object, which calls the `hourly_parse` function again, for the next URL we want to go to. This mechanism is built with a simple series of IF conditions, and a few other functions which support the shifting of days, months, and years ahead as needed. Here are the "support" functions we use here:

```
def leap_chk(n): # our leap-year checker
    if n % 4 > 0:
        return 28
    else:
        if n % 100 == 0:
            if n % 400 > 0:
                return 28
            else:
                return 29
        else:
            return 29

def lastday(y,m,d,index): # checks if given d is last day of given m
    if m == 2:
        if d == leap_chk(y): return True
        else: return False
    else:
        if d == index[m]: return True # index is days_dict output
        else: return False

def next_da(n): # gives str of next day
```

```
    if n < 9:
        return "0"+str(n+1)
    else:
        return str(n+1)

def next_mo(n,start_mo,last_mo): # gives str of next month
    if n == last_mo:             # these are both ints.
        return start_mo          # this guy is a string already.
    else:
        if n >= 9 and n < 12:
            return str(n+1)
        else:
            nexmo = (n+1)%12
            return "0"+str(nexmo)

def next_yr(y,m,last_mo): # assumes last day of month, moves +1 or +0.
    if m == last_mo:      # both ints.
        return str(y+1)
    else:
        return str(y)
```

and the recursive bit is as follows (will discuss below):

```
        # SEC 2: recursive scraping.

        if lastday(url_year,url_month,url_day,self.days_index):

            if url_year == self.lim_yr and url_month == self.lim_mo:

                list_num = (self.orderlist.index(url_region+"-"+url_site))

                if list_num == (len(self.orderlist)-1):
                    yield None
                else:
                    next_region = (self.orderlist[list_num+1])[0:2]
                    next_site = (self.orderlist[list_num+1])[3:]

                    yield Request(str(url_b1+url_timescale+url_b2+
                                      next_region+url_b3+next_site+url_b4+
                                      self.start_yr+url_b5+
                                      self.start_mo+url_b6+
                                      self.start_da+url_b7),
                                      callback=self.hourly_parse)
            else:
                yield Request(str(url_b1+url_timescale+url_b2+
                                  url_region+url_b3+url_site+url_b4+
                                  next_yr(url_year,url_month,self.lim_mo)+
                                  url_b5+
                                  next_mo(url_month,self.start_mo,self.lim_mo)+
                                  url_b6+
                                  "01"+url_b7),callback=self.hourly_parse)


        else: # note how we use url_month - 1 to keep month static here
            yield Request(str(url_b1+url_timescale+url_b2+
                              url_region+url_b3+url_site+url_b4+
                              str(url_year)+url_b5+
                              next_mo((url_month-1),self.start_mo,self.lim_mo)+
                              url_b6+
                              next_da(url_day)+url_b7),
                              callback=self.hourly_parse)
```

Every time the parser is given a URL, it checks the date. The date, using our checker function is either the last day of the month, or it is not. This binary set of alternatives gives us the first split. If not the last day of the month, everything but the day stays still.

If it is the last day of the month, we then become interest in, given that it is the last day of the month, whether or not the date is the "ending" year AND month that we specified in the initiator method. It's important to note how this works. Say we set the start date to be `1988/01/01` and want it to run until `2013/01`, i.e., until the end of January 2013. Then our cutoff month will be *January*, and we will get all the January entries for each year from 1988 to 2013, but will not get data for any other months. The algorithm is built such that by specifying start and end months, we can set a fixed period of months for which we collect data each year. If we only want some seasonal data or something to that end, this is a very natural structure. If both of those conditions are not satisfied, then we just move forward in the calendar to the next month (be it the end of a non-limit year or not), quite smooth.

If both are satisfied, however, we check where we are in the "orders" list we made, and assign that to `list_num`. If it is the last index value, then we have nothing to do since that would mean we had covered the designated time span for the final site/region pair, and would yield `None`. Should it not be the last index value, then we have more sites to scan for, and thus we "bump" the program up one space in the list, and make a `Request` using the start dates we initialized and the updated region/site IDs. That's all there is to it.

With a working spider in hand, we can effectively jump across all the sites we want, but with no output. To get output in a customizable way, Scrapy's "item pipelines" are ideal. We discuss that next.

Item pipeline

In the Scrapy documentation, items are often referred to as "containers" for the data our spiders scrape. That is an intuitive analogy, and item pipelines can be considered aptly named, since they are the mechanism in Scrapy which takes these containers sitting chock-full of data, and subsequently outputs that data into some more readily-consumed form (as designed by the programmer).

Initial setup In the `../table_scrape` folder, we will need to edit both `pipelines.py` and `settings.py`. It is very easy to forget about adjusting the settings file, so even before getting started on the pipelines file (we will be there for a while), it is advisable to select a name (or names) for the item pipeline(s) we intend to use and add the following line to the settings file:

```
ITEM_PIPELINES = ["table_scrape.pipelines.hourlyPipeline"]
```

where in this case we have just a single pipeline called `hourlyPipeline`. Unless we need to add more pipelines (we will not here), then our business with `settings.py` is now finished.

The Scrapy documentation has a fairly straightforward discussion with examples of how pipelines are structured, but in principle they are truly very simple. We have just one spider, and will be discussing the case of just one pipeline, but each pipeline behaves the same way and so the discussion extends naturally to the multiple-pipeline case.

A pipeline is a class which is initiated at the start of a crawl just once, and which uses the `process_item` method to perform some kind of action with *each data "entry,"* which is to say each individual *row* in the case of our table-scraping example here, and returns either that item having performed the action, or "drops" it from the flow of data coming in using the `DropItem` exception. We can also optionally use `open_spider` and/or `close_spider` which carries out some actions in a similar way when our spider opens/closes. With a single spider, `__init__` should work just as well, but depending on the problem, some action at the time of closure might be necessary. Here we will not need it.

Our pipeline works essentially "in parallel" with the algorithm we denoted in our parse function earlier, since `process_item` will be called for every single entry (in our case a dictionary with seven keys with unary values apiece each time) made via `hourlyItems`. What this means is that if our spider is working perfectly, it will perform its entire crawl when ordered to, regardless of what is going on in the pipelines file (aside from syntax errors halting the interpreter). As such, if our pipeline code is bad, we will get bad output even if the spider functions flawlessly, so our objective is to effectively "sync up" the pipeline functions with our spider's crawling and scraping.

Let us recall precisely what we said we wanted to output:

- Tabular data output (here we use CSV).
- One file per year of data, per region/site pair.
- Appropriate nomenclature for each file reflecting timescale, year, and region/site.

There is minimal information online regarding how to generate multiple files at intervals determined endogenously by the program using Scrapy, and a likely reason for that is the fact that Scrapy's item pipelines are perfectly capable of generating multiple files functionality-wise, and thus it really just becomes a matter of programming savvy. The author, not being a programmer, makes no claims whatsoever about the efficiency nor aesthetic appeal of the code presented here, apart from the fact that the mechanism used can be understood easily and that given the very general problem discussed above, it works in practice.

Initialization settings From the top of the `pipelines.py` file, we import a couple modules, and insert one function from the spider as well as another new one:

```
import csv
import items

def leap_chk(n): # our leap-year checker
    if n % 4 > 0:
        return 28
    else:
        if n % 100 == 0:
            if n % 400 > 0:
                return 28
            else:
                return 29
        else:
            return 29

def last_day(y,m): # special for the pipeline, gives last day of month.
    if m == "02":
        return str(leap_chk(int(y)))
    else:
        if m==4 or m==6 or m==9 or m==11:
            return str(30)
        else:
            return str(31)
```

where the `last_day` function will come in handy shortly, allowing our Pipeline algorithm to determine dynamically whether or not a given day is the last day of the given month and year (assuming our cut-off will always be the end of the month). We then create our class and put together the pre-run settings:

```
class hourlyPipeline(object):

    def __init__(self):
        # Pre-run settings (Pipeline)
        self.start_yr = "1988" # Str, 4 chars. Spider AND pipeline.
        self.lim_yr = "2012"   # Str, 4 chars. Spider AND pipeline.
        self.lim_mo = "12"     # Str, 2 chars. Spider AND pipeline.
        self.signal = 0
        self.index = 0
        self.orderlist = []

        with open("orders_up.csv",mode="r") as csvfile:
            self.orders = csv.reader(csvfile)
            for i in self.orders:
                a = (i[0])[0:2]
                b = (i[0])[3:]
                self.orderlist.append([a,b])

        self.limit = len(self.orderlist)

        # Initialize CSV for first year of first reg/site.
        self.hourlyCSV = csv.writer(open("h_r"+
                                    (self.orderlist[self.index])[0]+
                                    "_s"+(self.orderlist[self.index])[1]+
                                    "_"+self.start_yr+".csv","wb"))
        self.hourlyCSV.writerow(["reg_site","date","time","data_a","data_b",
                                 "data_c","data_d"])
```

The start/end dates are similar to the spider's code, though we only need a start year here. The `signal` will be used below. We again here initialize an `orderlist` once, since the pipeline will need to process things in *exactly* the same way as the spider, thus we provide our pipeline the same list (with slight tweaks to the format for convenience). The `index` will be a variable used to control how far our pipeline has moved along in the order list. **It should be noted:** this may seem overly complicated, and the reason for that is because we want to output *multiple* files. Were we content just churning out a single file with a preset format determined at the pipeline's initialization with all the content our spider scrapes, the code would be far, far simpler. That said, all of the data for even one site would likely have a hard time fitting on an Excel sheet, so multiple file creation is an unavoidable task here.

The first initialization of a file should illustrate how we will use the `csv` module and variable file name elements. The format will be `h_rXX_sXXXX_YYYY`, where the `h` stands for hourly (assumed throughout this document), and the remaining variables are region ID, site ID, and year respectively. One great thing about `csv` is that it lets us dictate the *order* of our output file columns, something that cannot be done with the very rudimentary default "feed export" option in Scrapy. We've opened the file for our first region/site's first year, and have written in the column headers.

Our sequential file-creation algorithm The basic task we need our pipeline to do (in addition to its basic function as an output generator) is to *keep track* of "where" it is in the long, long sequence of recording to be done, both date/time-wise and region/site-wise. Since initial settings are largely input by hand, it is easy to ensure that both the pipeline and the spider start at the same place, we need to ensure that the pipeline "responds" whenever we hit the end of a year so that it makes a new file just as the spider goes on to the next year, and similarly for the case of when our spider makes the jump from one site to another. The code is fairly short, though we will spend some time explaining it:

```
def process_item(self, item, spider): # runs for each data entry
    if self.signal > 0:
        if self.signal == 1:
            self.hourlyCSV = csv.writer(open("h_r"+
                             (self.orderlist[self.index])[0]+
                             "_s"+(self.orderlist[self.index])[1]+
                             "_"+str(int(self.year)+1)+".csv","wb"))
            self.hourlyCSV.writerow(["reg_site","date","time","data_a",
                                     "data_b","data_c","data_d"])
            self.signal = 0
        else:
            self.hourlyCSV = csv.writer(open("h_r"+
                             (self.orderlist[self.index])[0]+"_s"+
                             (self.orderlist[self.index])[1]+"_"
                             +self.start_yr+".csv","wb"))
            self.hourlyCSV.writerow(["reg_site","date","time","data_a",
                                     "data_b","data_c","data_d"])
            self.signal = 0

    self.year = item["date"][0:4]
    self.month = item["date"][5:7]
    self.day = item["date"][8:10]
    self.time = item["time"]
```

```
        # condition here just to make text readable; in code, within IF.
        cond_1 = self.day == last_day(str(self.year),str(self.month))

        if self.month==self.lim_mo and cond_1 == True and self.time=="24":
            self.signal = 1
            if self.index < self.limit and self.year == self.lim_yr:
                self.signal = 2
                self.index += 1

        self.hourlyCSV.writerow([item["reg_site"],item["date"],item["time"],
                                item["data_a"],item["data_b"],item["data_c"],
                                item["data_d"]])

        return item
```

By the way item pipelines are defined, it should really be quite easy for them to remain "aligned" with spiders, since at every iteration they have access to the most recent entry/row of data the spider has scraped, and thus always knows exactly what the spider knows. We just need to make sure it *uses* that data to tell itself to make a new file when certain limits/triggers are reached.

The basic mechanism is very simple: the `hourlyCSV` variable will be regularly updated by creating a new file whenever a certain "signal" is set off. Each time `process_item` is called, it can *only* read and write the "current" entry (seven values here) given by where the spider presently is scraping. It will happily continue writing line after line (using `csv`) until it reaches a trigger point, which will be the last entry (time is on 24-hour clock) of the last day of the specified ending month.

A short aside: It should be noted that Scrapy has a very handy feature in that when we use the XPath expression `select("td[2]/text()").extract()` to scrape text data, Scrapy gives it to us as a list. In the spider code we introduced above, since we wanted to deal with dashes, each Scraped datum was "put through the ringer" as it were, and entered into the dictionary as a string, not a list. Were one *not* to use a function such as `slago()` and insert the XPath result directly into the dictionary, we would need to add square brackets with a list index value (usually zero) each time we called `item["key"]` making it `item["key"][0]` so as to get just the text we want, without the character data it is wrapped in (`[u'XXX']`) by default.

The date and time currently being accessed by the spider is checked at each iteration using `year`, `month`, `day`, and `time`. We check those each round to see if we have reached a trigger point. When we do, we set global variable `signal` to `1`, and do one further check before writing the current entry, which must be the last entry to go into the currently open file. Once we return `item`, the spider will go to its next entry to scrape, which is why we turn on the signal ahead of time. The additional check to be made has two conditions: that we are *not* at the last region/site in the order list, and that we *are* at the last year of observation for that given site. In other words, it is the check which sees whether we move one year forward (in terms of file creation/naming) or whether we move one region/site forward and start at the initial year for that site.

One might be concerned that doing `+1` to the index for the last item in the set would result in an index value which is too large, since using a strictly less-than condition. While it looks like a rookie mistake, it is fine to let it pass the second set of conditions in the last case, because after we return, the spider will necessarily have finished its crawl, and thus the pipeline's `process_item` will not be called again.

When the signal is set to 1, we have a “next year” case. If set to 2, we have a “next region/site” case, and the newly-created file names reflect that.

In closing

With that, our spider and its pipelines have been built, and all the necessary tweaking has been completed. All that is left to do is to populate our orders list with a column of region/site pairs to check, and then enter

```
scrapy crawl table_scrape_spider
```

to set the machine into motion. With so much data, it is assuredly a long process even for a computer, but the peace of mind we get knowing that we can relax with a coffee while the machine works, and that we need not be concerned with errors made in transferring the data is assuredly worth both the wait, and the effort put into building the program.

References

[1] Holland, M.J. “Scrapy setup on 64-bit Windows.” Accessible online at http://feedbackward.com/content/scrapy_install.pdf

[2] Official Scrapy documentation. http://doc.scrapy.org/en/latest/index.html

[3] Official Scrapy tutorial. http://doc.scrapy.org/en/latest/intro/tutorial.html