

Creamos una potente araña web en sencillos pasos

SCRAPY DOO

© JasekS - Fotolia.com

Las arañas web son una de las bases de Internet, gracias a ellas miles de buscadores obtienen los datos que necesitan diariamente ¿Qué sería de Google sin su existencia? **POR JOSÉ MARÍA RUÍZ**

La librería Scrapy [1] representa a las arañas web lo que Django [2] a las web CGI: un completo y potente framework integrado que nos permitirá la creación de soluciones muy potentes de forma sencilla.

Para comprender Scrapy hay que entender toda una filosofía de trabajo que a día de hoy se está imponiendo en numerosos proyectos. Atrás quedan los años en los que una simple librería representaba una increíble ayuda en el desarrollo de un programa. Todos vivimos bajo presión ahora mismo, se nos exige que el proyecto esté listo para ayer, o simplemente no disponemos del tiempo necesario cuando lo hacemos por libre. Los programas ya no son programas, ahora exigimos que sean plataformas que nos permitan realizar innumerables tareas en paralelo y encima exigimos que sean fáciles de mantener, modulares y seguras.

Por eso el concepto de librería está perdiendo fuelle con tanta rapidez, y cada vez más proyectos adoptan un enfoque de framework de desarrollo. Me refiero a un conjunto de librerías, herramientas y conceptos que representan una filosofía de trabajo. Pensemos en el famoso Django, llamar librería a Django sería casi un insulto, puesto que en realidad representa un manera de ver el desarrollo web: nos ofrece todo lo necesario. Además automatiza casi todas las tareas que anteriormente hacíamos (con bastante desgana) a mano.

Pues bien, Scrapy copia casi milimétricamente la filosofía de Django (de hecho se

siente orgulloso de ello [3]), pero aplicándola al desarrollo de arañas web..

Pero, ¿Qué Es una Araña Web?

En un primer momento podemos pensar en una araña web como un navegador automatizado: entra una web, localiza los enlaces e información interesante, hace algo con ella y probablemente sigue alguno de los enlaces hasta una nueva web.

Dicho de esta manera, prácticamente cualquier solución nos vale. De hecho, Python dispone de algunas librerías, como urllib y urllib2, que podrían emplearse perfectamente para crear una araña web.

Pero, a poco que comencemos a tratar este asunto seriamente, comprenderemos que nos enfrentamos a un problema bastante complicado. Si miramos la Figura 1 comprobaremos lo sofisticado que es Scrapy. Scrapy considera que el proceso de

extracción de datos de una web se desarrolla en etapas. Primero podemos acceder a las webs, después movernos por el HTML mediante alguno de los sistemas que nos ofrece, recoger la información, pasarla por una tubería en la que puede sufrir diferentes cambios y generar notificaciones, y finalmente los datos se procesan pasando por una «tubería» donde distintos módulos trabajan con ellos. En cada paso del proceso Scrapy nos da libertad para emplear algunos de los módulos que nos ofrece o usar otros diferentes.

Creemos un Proyecto Scrapy

Pasemos a la acción, comencemos por crear un proyecto. Al igual que en Django, Scrapy nos ofrece una herramienta de administración que automatiza muchas acciones que anteriormente hacíamos a mano. Esta herramienta se llama *scrapy-ctl.py*, y deberíamos poder acceder a ella una vez tenga-

Listado 1: lm_spider.py Inicial

```
01 # -*- coding: utf-8 -*-           10 ]
02                                     11
03 from scrapy.spider import          12 def parse(self, response):
   BaseSpider                         13     hxs =
04 from scrapy.selector import        14     HtmlXPathSelector(response)
   HtmlXPathSelector                  15     titulo =
05                                     16     hxs.select('/html/head/title/t
06 class LmSpider(BaseSpider):         17     ext()').extract()
07     domain_name =                   18     self.log(u"El título es
   "linux-magazine.es"                {0}").format(titulo[0])
08     start_urls = [                  19
09     "http://www.linux-magazine.es/  20 SPIDER = LmSpider()
```

Listado 2: parse()

```

01 def parse(self, response):
02     hxs =
03     HtmlXPathSelector(response)
04     revistas =
05     hxs.select('/html/body/table[
06     1]/tr/td[2]/table[1]/tr/td[1]
07     /table[1]/tr/td[1]/table[1]/t
08     r')
09     enlace_numero = [
10     urllib.basejoin('http://www.l
11     inux-magazine.es',
12     revista.select('td[2]/a[2]/@h
13     ref').extract()[0]) ]
14     portada =
15     revista.select('td[3]/text()'
16     ).extract()
17     dvd =
18     revista.select('td[4]/text()'
19     ).extract()
20     self.log(u"{0} =>
21     {1}".format(numero[0],
22     portada[0]))

```

mos instalado Scrapy (por ejemplo mediante nuestro sistema de paquetes). Es un comando, por lo que podremos invocarlo desde nuestra shell y ver qué opciones nos proporciona.

```

$ scrapy-ctl.py
scrapy-ctl.py
Scrapy 0.8 - no active project

Usage
=====

To run a command:
scrapy-ctl.py <command>
[options] [args]

To get help:
scrapy-ctl.py <command> -h
....

```

Los comandos que nos proporciona Scrapy son:

- fetch [opciones] <url >
- runspider [opciones] < spider_file >
- settings [opciones]
- shell [url|fichero]
- startproject < nombre_del_proyecto >

Al igual que en Django, podemos arrancar un shell con Scrapy, que no será otra cosa sino una instancia de IPython (ver Recurso 4) con Scrapy precargado, como veremos más adelante. Pero el comando que nos interesa ahora mismo es *startproject*:

```

$ scrapy-ctl.py startproject >
linuxmag $

```

¿No ha pasado nada? Siguiendo la más añeja tradición Unix, *scrapy-ctl.py* sólo emite información en caso de error, por lo que todo ha ido bien. Dispondremos ahora

de un nuevo directorio llamado *linuxmag* que contiene todo lo necesario para crear nuestra araña.

```

$ cd linuxmag
$ ls
linuxmag/ scrapy-ctl.py

```

Por tanto, existe un directorio que, de nuevo, se llama *linuxmag* y una copia de *scrapy-ctl.py*. Exploremos más.

```

$ cd linuxmag
$ ls
__init__.py __init__.pyo
items.py pipelines.py
settings.py spiders/

```

Scrapy ha copiado de Django hasta la arquitectura interna, crea una librería añadiendo un fichero *__init__.py* vacío y genera varios ficheros, cada uno de los cuales representa una parte de proceso. Además dispone de un fichero *settings.py*, donde podremos configurar detalles como por ejemplo cómo se identificará Scrapy cuando acceda a una web.

Mi Primera Web Escaneada

Como ejemplo, vamos a escanear la web de Linux Magazine para extraer la información de los números que hay publicados online. Supongo que sabrás que Linux Magazine libera sus artículos en PDF en la Web ¿no?

Veamos, Linux Magazine dispone de una web donde aparece la relación de números online: <http://www.linux-magazine.es/Magazine/Archive>. Lo primero que haremos es definir un *spider* (araña) en el directorio *spiders*. Creamos un fichero con el nombre *lm_spider.py* con el contenido del Listado 1. Si vamos al directorio *linuxmag*,

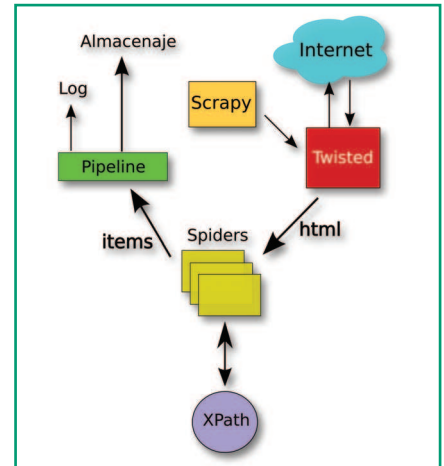


Figura 1: Esquema de funcionamiento.

donde está la copia de *scrapy-ctl.py*, podemos ejecutar:

```

$ ./scrapy-ctl.py crawl >
linux-magazine.es
...
2010-04-07 04:27:20+0100 [-]
scrapy.management.telnet.TelnetCon
sole starting on 6023
2010-04-07 04:27:20+0100 [-]
scrapy.management.web.WebConsole
starting on 6080
...
2010-04-07 04:27:21+0100
[linux-magazine.es] DEBUG: El
título es Linux Magazine -
Archivo En Línea: Tod....
...
$

```

Scrapy genera una gran cantidad de información de log cuando se ejecuta. En el Listado 1 encontramos la clase *LmSpider* que hereda de *BaseSpider*. La configuramos indicándole de qué dominio será responsable y en qué urls arrancaremos el spider. Por defecto, Scrapy pasará un objeto *response* con los datos de la página al método *parse()*, por lo que hemos definido ese método. Aparece ahora uno de los elementos más importantes de Scrapy: *HtmlXPathSelector()*. Esta clase nos permite usar el lenguaje *XPath* para navegar dentro del HTML buscando la información que necesitamos. *XPath* trata el documento HTML como si fuese un árbol, muy parecido a una ruta del sistema de ficheros. Por eso usamos la ruta *"/html/head/title/text()"* para acceder al título de la página mediante el método *select()*, y con *extract()* obtenemos el contenido del texto del título. Por último, usamos el método *log* de *BaseSpider* para

Listado 3: Fichero items.py

```

01 from scrapy.item import
    Item, Field
02
03 class LmItem(Item):
04     numero = Field()
05     enlace_numero = Field()
06     portada = Field()
07     dvd = Field()

```

sacar esta información en el shell. Si observamos arriba el resultado de la ejecución, veremos que aparece el texto con el título de la página.

Para profundizar en XPath lo mejor es pasarse por [5], pues es un lenguaje muy potente y demasiado extenso para tratarlo en este artículo.

Pero hay otros detalles interesantes en el log. Como la información que genera Scrapy es muy abundante, he resaltado dos líneas que nos indican que Scrapy arranca tanto un servidor web como uno telnet. Scrapy ha sido diseñado pensando en que se ejecutará durante mucho tiempo, y estos dos interfaces (web y telnet) nos permiten recabar información y controlar una aplicación Scrapy en ejecución de forma remota. Como ya he dicho antes, Scrapy es muy avanzado y potente. Si quieres sacar partido de estos dos servidores, pásate por el Recurso 1 y estudia la documentación de Scrapy, trae ejemplos muy buenos.

Con nuestro primer contacto con Scrapy a las espaldas, podemos pasar ahora a algo más complicado.

Trasteando con el Shell

El problema principal al escanear una web es dar con la expresión *XPath* que extraiga

los datos que necesitamos. El HTML que te puedes encontrar en la Web puede ser de lo más diverso, y algunas webs son especialmente complejas. Cambiar el código del spider, ejecutarlo, comprobar los datos, volver a cambiar... puede convertirse en un proceso largo y tortuoso.

Scrapy nos ofrece una solución para poder trastear con la web de forma más sencilla. Si ejecutamos el siguiente comando:

```

$ scrapy-ctl.py shell >
http://www.linux-magazine.es/ >
Magazine/Archive
...
Fetching <GET
http://www.linux-magazine.es/
Magazine/Archive>...
Available objects
=====

xss      : <XmlXPathSelector
(http://www.linux-magazine.es/
Magazine/Archive) xpath=None>
url      :
http://www.linux-magazine.es/
Magazine/Archive
request  : <GET
http://www.linux-magazine.es/
Magazine/Archive>
spider   : None
response : <200
http://www.linux-magazine.es/
Magazine/Archive>
hxs      : <HtmlXPathSelector
(http://www.linux-magazine.es/
Magazine/Archive) xpath=None>
item     : Item()

```

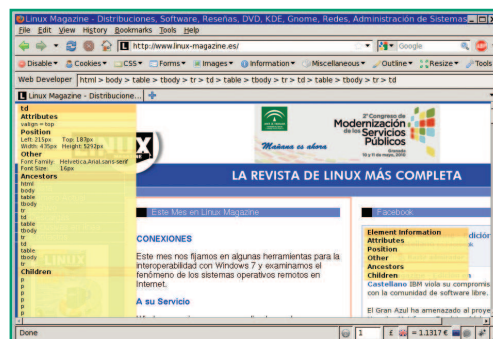


Figura 2: Webdeveloper en acción.

```

....
In [1]:

```

dispondremos de un shell de IPython listo para hacerlo. Tiene abierta una conexión con la página en la que estamos trabajando, por lo que podemos ponernos a trastear inmediatamente.

```

In [2]: hxs.select >
('/html/head/title/text()')
Out[2]: [<HtmlXPathSelector >
(text) xpath=/html/head/title/
text()]
In [3]: hxs.select >
('/html/head/title/text()').>
extract()
Out[3]: [u'Linux Magazine - >
Archivo En L\xedxednea: Todos ...']

```

Es recomendable usar alguna herramienta accesoria para hacer esta búsqueda de forma más «amigable» que la prueba y error. A mí me gusta especialmente el plugin de Firefox *webdeveloper*, ver [6], el cual posee en el menú *Information* la opción “Display Element Information” que

Listado 4: Nuevo lm_spider.py con Items

```

01 # -*- coding: utf-8 -*-
02
03 from scrapy.spider import
    BaseSpider
04 from scrapy.selector import
    HtmlXPathSelector
05 from linuxmag.items import
    LmItem
06 import urllib
07
08 class LmSpider(BaseSpider):
09     domain_name =
        "linux-magazine.es"
10     start_urls = [
11         "http://www.linux-magazine.es
        /Magazine/Archive",
12     ]
13
14     def parse(self, response):
15         hxs =
            HtmlXPathSelector(response)
16
17         revistas =
18             hxs.select('/html/body/table[
19                 1]/tr/td[2]/table[1]/tr/td[1]
20                 /table[1]/tr/td[1]/table[1]/t
21                 r')
22         items = []
23         for revista in revistas:
24             numero =
25                 revista.select('td[2]/a[2]/te
26                 xt()').re('\w+\s+([0-9]+)')
27             if
28                 revista.select('td[2]/a[2]/@h
29                 ref'):
30                 item = LmItem()
31                 item['numero'] = numero
32                 item['enlace_numero'] = [
33                     urllib.basejoin('http://www.l
34                     inux-magazine.es',
35                     revista.select('td[2]/a[2]/@h
36                     ref').extract()[0]]
37                 item['portada'] =
38                     revista.select('td[3]/text()')
39                     .extract()
40                 item['dvd'] =
41                     revista.select('td[4]/text()')
42                     .extract()
43                 items.append(item)
44         return items
45
46 SPIDER = LmSpider()

```

Consigue



3X1

por sólo

6,95€

y además ...
con cada
número de
regalo un
fantástico
doble DVD
100% Linux

¡ Pídelos ya !

www.linuxmagazine.es/prueba

Listado 5: Fichero pipeline.py

```

01 import json
02
03 class
LinuxmagPipeline(object):
04     def process_item(self,
domain, item):
05         return item
06
07 class
JsonWriterPipeline(object):
08     def __init__(self):
09         self.fichero =
open('revistas.json', 'wb')
10
11     def process_item(self,
spider, item):
12         datos = {'numero' :
item['numero'],
13                 'enlace' :
item['enlace_numero'],
14                 'portada':
item['portada'],
15                 'dvd' : ['dvd']}
16         json.dump(datos,
self.fichero)
17
18     return item

```

nos permite ver la ruta de cualquier elemento de una web con sólo pulsarlo, ver Figura 2.

Investigando un poco llegué a la ruta:

```

'/html/body/table[1]/tr/
td[2]/table[1]/tr/td[1]/
table[1]/tr/td[1]/
table[1]/tr'

```

Hay que tener en cuenta que en *XPath* debe especificarse el número de un elemento si aparece repetido varias veces. Con esta ruta podemos extraer la información de los números de la revista cambiando el método *parse()* por el que aparece en el Listado 2. Si volvemos a ejecutar, podremos ver en el log:

```

$ ./scrapy-ctl.py crawl
linux-magazine.es
....
2010-04-07 05:10:11+0100
[linux-magazine.es] DEBUG: 16 =>
Tema de Portada: Voz Sobre IP
2010-04-07 05:10:11+0100
[linux-magazine.es] DEBUG: 15 =>
Tema de Portada: Cazadores de
Virus
2010-04-07 05:10:11+0100
[linux-magazine.es] DEBUG: 14 =>
Tema de Portada: Fotografía
Digital
2010-04-07 05:10:11+0100
[linux-magazine.es] DEBUG: 13 =>
Tema de Portada: Juegos
....

```

Si observas el Listado 2, te habrá llamado la atención la línea:

```

numero = revista.select
('td[2]/a[2]/text()').
re('\w+\s+([0-9]+)')

```

Una vez que tenemos un objeto *XPath*, podemos consultar su interior empleando de nuevo *XPath*, y posteriormente usar *extract()* para obtener una cadena de texto con su contenido. Pero en este caso hemos empleado el método *re()*, el cual nos permite extraer datos utilizando una expresión regular. Como sólo queremos el número de la revista, hemos usado la expresión regular *'([0-9]+)'*. Tanto *extract()* como *re()* devuelven una lista.

¡Ya tenemos los datos de las revistas! Ahora necesitaríamos almacenarlos en algún formato.

La Tubería

Cuando se recuperan datos en el método *parser()*, el objetivo no es sacarlos por el log, sino devolverlos dentro de un item. Si observamos el contenido del directorio *linuxmag*, encontraremos un fichero llamado *items.py* que contiene algo de código. Scrapy nos ofrece un sencillo sistema para almacenar los datos recogidos para su posterior tratamiento, en el Listado 3 aparece el contenido de nuestro fichero *items.py* y en el Listado 4 el nuevo método *parse()*. Ejecutamos de nuevo:

```

$ ./scrapy-ctl.py crawl
linux-magazine.es
...
2010-04-07 05:19:06+0100
[linux-magazine.es] INFO: Passed
LmItem(portada=[u'Tema de Portada:
Redes'], dvd=[u'\n', u'\n'],
enlace_numero=[u'http://www.linux
-magazine.es/issue/01'],
numero=[u'01'])
...

```

Ahora Scrapy nos informa de que está creando objetos *LmItem*, pero los está devolviendo y nadie los está esperando. Debe-

mos añadir un procesador a la tubería de procesamiento. Veamos el código del Listado 5.

En él definimos la clase *JsonWriterPipeline* que posee un método *process_item()*. La razón por la que se llama tubería (“pipeline”) se debe a que todo lo que entra por una tubería debe salir de ella, así que el método *process_item()* devuelve el item que recibe para que otro elemento de la tubería pudiese hacer algo con él.

En nuestro caso, lo que hacemos es usar el formato *Json*, para el que Python posee una librería en la instalación por defecto, y almacenamos los datos que hemos recopilado en un fichero llamado *revistas.json*. Este formato es muy sencillo de procesar posteriormente.

Pero nos falta algo, ¿recuerdas el fichero *settings.py*? Para activar un elemento de la tubería debemos registrarlo allí. Sólo tenemos que añadir la siguiente línea en su interior:

```

ITEM_PIPELINES =
['linuxmag.pipelines.
JsonWriterPipeline']

```

¡Y listo! Cuando volvamos a ejecutar el spider aparecerá el fichero con los datos en *Json*.

Conclusión

Los spiders son uno de los componentes más importantes de la red, sin ellos, empresas de búsqueda como Google nunca habrían existido. Scrapy representa un paso hacia adelante en el mundo de los spiders, aportando modularidad, una estructura predefinida y herramientas que nos hacen la vida mucho más fácil en el complicado mundo de la Web. Sólo hemos visto una mínima parte de sus posibilidades en este artículo, animo al lector a profundizar sobre Scrapy en el Recurso 1. ■

RECURSOS

- [1] Scrapy: <http://scrapy.org/>
- [2] Django: <http://http://www.djangoproject.com/>
- [3] Faq de Scrapy: <http://doc.scrapy.org/faq.html>
- [4] IPython, el super shell de Python: <http://ipython.scipy.org/>
- [5] XPath: <http://es.wikipedia.org/wiki/XPath>
- [6] Webdeveloper para Firefox: <https://addons.mozilla.org/en-US/firefox/addon/60>