Escarbando a lo loco

ExactasPrograma - Datos

Facultad de Ciencias Exactas y Naturales, UBA

Invierno 2020

¿Qué vamos a hacer hoy?

- Nos peleamos para **instalar** un paquete, la mayoría ganó, pero estoy seguro que más de uno todavía está en la lucha.
- Ya aprendimos cómo bajar, **manualmente**, algunas páginas.
- Vimos cómo **graficar la frecuencia** de aparición con una nube de palabras.
- Escarbamos un poco sobre cómo hacer para **filtrar palabras** o cadenas de caracteres que no me interesan (números por ejemplo).
- Hoy vamos a ir un poco más allá. Vamos a juntar lo que vimos, junto con algunos otros condimentos.
- Al terminar todos los ejercicios, vamos a tener una aplicación completa para analizar artículos.
- Quizás, la policía de Internet nos odie un poco más después de hoy.

Importante

Vamos a mezclar la consola (terminal) para ejecutar las pruebas y un editor o Spyder para escribir nuestro código. ¡Las dos cosas son necesarias!

¿Cómo se organiza un programa para *crawlear* con `scrapy`?

- Ya dijimos que `scrapy` **no** es una aplicación, sino un **framework**.
- Esto significa que nos ayuda ofreciendo **ciertas** operaciones, pero nosotros tenemos que **programar** otras.
- `scrapy` se maneja con el concepto de **proyecto**. Todo lo que hacemos queda encerrado en un proyecto:

  ```
  scrapy startproject sherlock
  ```

  Se ejecuta desde la terminal de Linux o desde la Conda shell en Windows. Crea un esqueleto de proyecto vacío, pero con todos los archivos y configuración básica ya preparada.

  ```
  ./scrapy.cfg
  ./sherlock
  ./sherlock/__init__.py
  ./sherlock/items.py
  ./sherlock/middlewares.py
  ./sherlock/pipelines.py
  ./sherlock/settings.py
  ./sherlock/spiders
  ./sherlock/spiders/__init__.py
  ```

Un pasito más

- Con el comando anterior, tenemos el esqueleto preparado, ahora tenemos que agregar un programa que **haga algo**.
- Dentro del mundo `scrapy`, el encargado de *trabajar* es el `spider`.
- Un `spider` es un **programa** escrito en `Python` que tiene definidas algunas funciones que `scrapy` usa para hacer la descarga de material de Internet.
- Nuevamente, `scrapy` tiene un comando para crear un esqueleto para un `spider`:

```
scrapy genspider -t basic messi exactasprograma.exactas.uba.ar
```

El archivo `messi.py` que está en la carpeta `spiders` tiene este contenido:

```
import scrapy
class MessiSpider(scrapy.Spider):
    name = 'messi'
    allowed_domains = ['exactasprograma.exactas.uba.ar']
    start_urls = ['http://exactasprograma.exactas.uba.ar/']

    def parse(self, response):
        pass
```

Detalles del scrapy

El código que genera tiene algunos detalles que nos sirve para aprender un poco más de `Python`:

- `class MessiSpider(scrapy.Spider):`
- Esta línea me indica que nuestro `spider` va a **heredar** de la clase `scrapy.Spider`.
- Traducido significa: hay un montón de cosas que alguien ya programó y al hacer eso, no las tengo que ni pensar, están **disponibles**.
- Que estemos definiendo nuestro `spider` dentro de una clase te brinda muchas ventajas, pero tiene un costo: hay que **entender** un poco de qué se trata.

Clase y objeto

Conceptos mínimos de lo que necesitamos: **Clase** y **Objeto**.
Veamos ejemplos:

- Clase: `List`. Objetos: [1,2,3] o [’a’, ’b’, ’c’]
- La **clase** me define qué operaciones puedo hacer y qué información voy guardar.
- Los **objetos** son las distintas variables que tienen un valor concreto.
- **Clase**: Dataframe. **Objetos**: todos los `df` que fueron definiendo.
- Venimos usando los conceptos de clase y objeto desde el principio, solo que no lo tenían por ese nombre.
- Cada vez que declaramos una variable, estamos **instanciando** un objeto de esa clase. Esta es la jerga.
- Pero nosotros simplemente escribimos

```
a = ['a', 'b', 'c']
a.append('d')
print(a)
```

  `a` es una **instancia** de la clase **List**. Las operaciones que puedo usar sobre `a` están en la documentación de `list` (alguien ya las programó).
- La función `append` la podemos **aplicar** a `a` porque está definida como parte de la clase `list`.
- La función `getLotsOfMoney` **no la podemos usar** porque nadie la definió ni programó dentro de la clase `list`.

Sigamos con los detalles del `spider`

```
class MessiSpider(scrapy.Spider):
    name = 'messi'
    allowed_domains = ['exactasprograma.exactas.uba.ar']
    start_urls = ['http://exactasprograma.exactas.uba.ar/']
```

- En nuestro `spider` define tres variables: una es un `string` y las otras dos son listas de `string`.
- La variable `name` la necesita `scrapy` para identificar a nuestro `spider` (se podrían tener varios por proyecto, a nosotros nos va a alcanzar con uno). Debe ser **único** dentro de un proyecto.
- La variable `allowed_domains` la usa `scrapy` para **limitar los dominios** que revisa (solo se permite visitar los incluidos en esta lista). Cuando crea esta estructura básica, le pone el mismo sitio que pusimos en el comando.
- La variable `start_urls` es la dirección que usa `scrapy` para comenzar a buscar. Es la primera página que va a buscar (hace automáticamente el `fetch`).

¿Cómo lo corremos?

- Vamos a probar correr el `spider` sencillo que tenemos. Desde el directorio del proyecto, hacer:

```
scrapy crawl messi
```

- Esto hace que `scrapy` ponga a andar toda la maquinaria para traer las páginas que están indicadas en el spider.
- Realiza las siguientes acciones:
  1. En la jerga: instancia el spider messi (asigna una variable con `messi`).
  2. Le pide la *url* de comienzo, que está en la variable `start_urls`.
  3. Verifica que el dominio está dentro de los permitidos por la variable `allowed_domains` (esta variable puede no estar definida, significa que admite cualquier dominio).
  4. Arma todo lo necesario para hacer el `request` al servidor.
  5. Realiza el pedido y guarda el resultado en la variable `response`.
  6. Llama a la función `parse` para que procese el resultado.

¿Qué ves cuándo me ves?

- Al correr esto en la terminal, van a ver un montón de mensajes.
- Al principio, todos los de inicialización de `scrapy`.
- En el medio los de trabajo del `spider`.
- Al final muestra un resumen de lo realizado.

```
2020-08-28 10:21:19 [scrapy.core.engine] INFO: Spider opened
2020-08-28 10:21:19 [scrapy.extensions.logstats] INFO: Crawled 0
    pages (at 0 pages/min), scraped 0 items (at 0 items/min)
2020-08-28 10:21:19 [scrapy.extensions.telnet] INFO: Telnet console
    listening on 127.0.0.1:6023
2020-08-28 10:21:19 [scrapy.core.engine] DEBUG: Crawled (404) <GET
    http://exactasprograma.exactas.uba.ar/robots.txt> (referer:
    None)
2020-08-28 10:21:19 [scrapy.core.engine] DEBUG: Crawled (200) <GET
    http://exactasprograma.exactas.uba.ar/> (referer: None)
2020-08-28 10:21:19 [scrapy.core.engine] INFO: Closing spider (
    finished)
```

- Cuando vean un `200` como `status` es que todo fue **bien**.
- Cuando vean un `404` como `status` es que todo fue **mal**.

Pongamos un poco de código

Vamos a implementar la función `parse` para que imprima en pantalla y en el *log* de `scrapy` información de la página que trajo:

```
class MessiSpider(scrapy.Spider):
    name = 'messi'
    allowed_domains = ['exactasprograma.exactas.uba.ar']
    start_urls = ['http://exactasprograma.exactas.uba.ar/']

    def parse(self, response):
        print("El spider " + self.name + " trajo informacion desde la
    direccion " + str(response.ip_address))
        self.log("URL leido: " + response.url + " status=" + str(
    response.status))
        self.log("Comienzo de crawl: " + str(self.start_urls))
```

Importante

Notar que dentro de la función `parse` puede usar las variables que están definidas en la clase, para hacerlo tengo que indicarlo poniendo el `self` antes.

¿Y ahora qué vemos?

- La función que `scrapy` ejecuta cuando trae una página es:

```
def parse(self, response):
    print("El spider " + self.name + " trajo informacion desde la
    direccion " + str(response.ip_address))
    self.log("URL leido: " + response.url + " status=" + str(
    response.status))
    self.log("Comienzo de crawl: " + str(self.start_urls))
```

- En los mensajes que quedaron en la terminal, hay tres líneas distintas a la corrida anterior:

```
El spider messi trajo informacion desde la direccion 157.92.32.35
2020-08-28 10:43:11 [messi] DEBUG: URL leido:http://exactasprograma.
    exactas.uba.ar/ status=200
2020-08-28 10:43:11 [messi] DEBUG: Comienzo de crawl: ['http://
    exactasprograma.exactas.uba.ar/']
```

¡Y ahora vamos a recorrer otras páginas!

```
class MessiSpider(scrapy.Spider):
    name = 'messi'
    start_urls = ['http://exactasprograma.exactas.uba.ar/']
    libro = 1
    def parse(self, response):
        # Esta lista representa el resultado de un procesamiento
        mis_links = ['https://www.gutenberg.org/files/8117/8117-h/8117-h
.htm', 'https://www.gutenberg.org/files/36016/36016-h/36016-h.htm',
 'https://www.gutenberg.org/files/537/537-h/537-h.htm', 'https://
www.gutenberg.org/files/38982/38982-h/38982-h.htm']
        for link in mis_links:
            yield scrapy.Request(link, callback=self.parse_libro)

    def parse_libro(self, response):
        print("El spider " + self.name + " trajo el libro nro " + self.
libro + " desde la direccion " + str(response.ip_address))
        self.log("URL leido: " + response.url + " status=" + str(
response.status))
        self.libro+=1
```

- `yield` es un `return` **especial** que devuelve un valor, pero la función continúa ejecutando.
- Va a **generar** un `request` por cada elemento de la lista que tengo armada.
- Además, podemos indicarle que la página que nos trajimos, la **procese otra**

¿Y si corremos esto?

```
2020-08-28 12:26:29 [scrapy.core.engine] DEBUG: Crawled (404) <GET http
    ://exactasprograma.exactas.uba.ar/robots.txt> (referer: None)
2020-08-28 12:26:29 [scrapy.core.engine] DEBUG: Crawled (200) <GET http
    ://exactasprograma.exactas.uba.ar/> (referer: None)
2020-08-28 12:26:29 [scrapy.core.engine] DEBUG: Crawled (200) <GET https
    ://www.gutenberg.org/robots.txt> (referer: None)
2020-08-28 12:26:32 [scrapy.core.engine] DEBUG: Crawled (200) <GET https
    ://www.gutenberg.org/files/38982/38982-h/38982-h.htm> (referer:
    http://exactasprograma.exactas.uba.ar/)
El spider messi trajo el libro nro 1 desde la direccion 152.19.134.47
2020-08-28 12:26:32 [messi] DEBUG: URL leido: https://www.gutenberg.org/
    files/38982/38982-h/38982-h.htm status=200
2020-08-28 12:38:55 [scrapy.core.engine] DEBUG: Crawled (200) <GET https
    ://www.gutenberg.org/files/537/537-h/537-h.htm> (referer: http://
    exactasprograma.exactas.uba.ar/)
2020-08-28 12:38:55 [scrapy.core.engine] DEBUG: Crawled (200) <GET https
    ://www.gutenberg.org/files/36016/36016-h/36016-h.htm> (referer:
    http://exactasprograma.exactas.uba.ar/)
El spider messi trajo el libro nro 2 desde la direccion 152.19.134.47
```

Terminando

Importante

Con esta estructura podemos ir trayendo páginas, procesarlas sacando *links* de otras páginas a visitar y procesar específicamente de acuerdo a lo que esté buscando.

¡Ya pueden bajar la guía y empezar a trabajar!