El Poder de Gnuplot y las expresiones regulares

GRAFICAS CON PYTHON

Lo mejor de ser una lenguaje interpretado y dinámico es que puedes hablar con otras aplicaciones muy fácilmente y Python es capaz de hablar con Gnuplot. Ya podemos hacer gráficas con Python.

POR JOSÉ MARÍA RUIZ Y PEDRO ORANTES

Existen muchos programas que generan gráficas sobre información importante. Por ejemplo, muchos sitios web tienen gráficas que muestran cuantos accesos se han realizado ese mismo día. También sería interesante controlar la cantidad de accesos por SSH que nuestra máquina recibe como medida de seguridad.

Gráficas y más gráficas

Y ya que estamos, ¿por qué no hacer gráficas de cosas interesantes? Una de las cosas que más nos atraen es controlar como crece, o decrece, el número de ficheros en el sistema operativo o su tamaño. ¿Es útil? no lo sé, pero desde luego es curioso.

Nuestro objetivo este mes es el siguiente: crear un programa con Python que mediante expresiones regulares que localice entradas en un fichero de log, agrupe por fecha y genere un histograma con ellas. Parece complicado, pero vamos a ver como no lo es tanto.

Para ello vamos a utilizar las expresiones regulares que Python provee. Con ellas podremos procesar texto en busca de patrones fácilmente. Una vez que tengamos un conocimiento básico sobre ellas echaremos un vistazo a Gnuplot y a la manera de usarlo desde Python.

Las expresiones regulares

Las expresiones regulares trabajan con símbolos, en nuestro caso caracteres, y definen patrones de caracteres como, por ejemplo, palabras o direcciones IP. Podemos buscar una frase o parte de ella, por ejemplo "From: algo.tw". Esta expresión regular corresponde con … ¡ella misma!, lo cual no es demasiado útil. Necesitamos algo que abarque más posibilidades.

Lo interesante es definir patrones. Imaginemos que queremos definir un patrón para capturar direcciones de correo. Normalmente estará formada por:

- unas secuencia de caracteres pertenecientes a un conjunto restringido (por ejemplo no se permiten tildes)
- una arroba
- una secuencia de caracteres pertenecientes al mismo conjunto restringido
- una última secuencia de dominio

O sea, algo como "spamer@spam.tw". Si queremos crear un patrón para este tipo de cadena de símbolos necesitamos reglas.

Las expresiones regulares permiten imponer restricciones. Podemos imponer condiciones sobre la aparición de los símbolos. En concreto podemos determinar

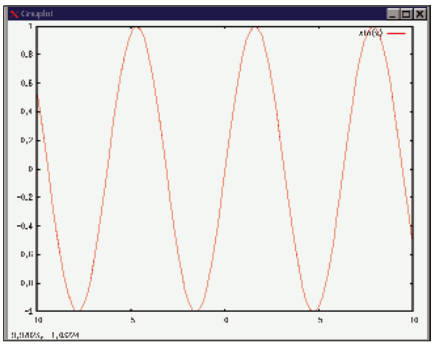

Figura 1: Gnuplot muestra representaciones de funciones de manera rápida y sencilla.

la cantidad y la posición. Indicar si ese carácter o caracteres van a aparecer o no y la cantidad de veces que lo hará.

El patrón más simples es que el busca la aparición obligatoria de un solo carácter. Basta con poner ese carácter tal cual en la expresión regular: "q". Si el carácter es uno de entre un conjunto de posibles caracteres, por ejemplo los caracteres "K" o "c", encerramos a ambos entre corchetes: "[Kc]". Esto indica que esperamos un solo carácter, pero escogido entre "K" o "c". En el caso de que ese único carácter sea opcional acompañaremos la expresión de un signo de interrogación: "[Kc]?".

Ahora vamos a ver como controlar la cantidad de caracteres. En nuestra introducción básica solo vamos a ver las reglas más simples. Estas son las que indican si el carácter puede aparecer seguido "0 o más veces" o "1 o más veces". En en primer caso acompañamos al carácter o caracteres (entre "[]") de un "*" y en el segundo caso de un " + ":

- [a]*: "", "a", "aa", "aaa"… *
- [a] + : "a", "aa", "aaa",…

Sería muy engorroso tener que poner todo el abecedario si queremos definir el patrón para una palabra. Por eso existen los grupos de caracteres. El grupo "a-z" define todas las letras del abecedario y "0-9" todos los números. Así, el patrón "[a-z]*" define una secuencia de 0 o más caracteres, y "[0-9] + " un número de al menos una cifra (que puede ser el "0"). El patrón "[A-Z]" se corresponde con las letras del abecedario en mayúsculas. Existe también patrones especiales que definen, por ejemplo, los caracteres que pueden formar una palabra, o los separadores.

Cuando un carácter puede ser mal interpretado, porque ya forma parte de la sintaxis de las expresiones regulares, entonces hay que acompañarlo por un "\". El caso típico es el carácter "-", que aparecerá como "\-".

Después de esta breve introducción, un correo electrónico que provenga de Taiwan podría corresponder con la siguiente expresión regular: "[\-a-zA-Z0-9] + @[\-a-zA-Z0-9] + .tw".

Imaginemos que queremos un patrón para un carácter que puede ser cualquiera símbolo menos el "9". Para ello usamos el carácter "^" dentro de "[ ]". "[^9]" sería el patrón que correspondería con cualquier carácter menos el "9".

Dentro de los corchetes "^" define un patrón inverso, decimos lo que no queremos y el patrón se corresponde con el resto. En cambio si usamos "^" fuera de los corchetes su significado cambia, pasa a significar "al principio". La expresión regular "^[0-9] + " corresponde con una cadena de números que está al principio de la cadena en la que buscamos, si no está al principio no se corresponderá. La expresión regular "[^0-9] + " corresponde con una cadena de al menos un símbolo que no puede ser numérico. Por tanto, hay que ser cuidadoso con donde se ponen los modificadores (ver Tabla de modificadores).

Con esto nos podemos hacer una idea de cómo son las expresiones regulares. En realidad faltan muchos más modificadores, pero el lector pueden encontrar gran cantidad de tutoriales sobre expresiones regulares en Internet. En particular recomendamos el indicado en el recurso [3], pero tiene el inconveniente de estar en inglés.

Expresiones Regulares en Python

En Python tenemos expresiones regulares a través del paquete *re* (de "Regular Expressions"). Este paquete forma parte del sistema base de Python por lo que nos será necesario instalarlo. Lo primero que vamos a hacer va a ser definir una expresión regular simple y juguetear con ella.

```
>>> import re
>>> p = re.compile("[ab]+")
>>> print p
<_sre.SRE_Pattern object➥
at 0x81418c0>
>>>
```

Hemos creado un objeto que representa la expresión regular "[ab]*", o sea, una secuencia de "a"y "b" que puede tener longitud 0. Como vemos al imprimir el contenido de "p" no es más que un objeto como todos los demás de Python.

Vamos a pasarle unas serie de cadenas a ver que nos dice:

```
>>> print p.match("34",1)
None
>>> print p.match("abaaab",1)
<_sre.SRE_Match object at➥
0x81b4800>
>>> print p.match("abaaabc",1)
<_sre.SRE_Match object at➥
0x81b4800>
>>>
```

En el primero ejemplo vemos como nos devuelve *None*. Eso significa que no existe ninguna subcadena dentro de la cadena que le hemos pasado que se corresponda con el patrón. En el segundo ejemplo vemos como nos devuelve un objeto, ese objeto contiene información sobre la correspondencia hallada. En este caso es toda la cadena pasada.

En el tercer ejemplo vuelve a devolvernos un objeto, pero vemos que esa "c" final no está en el patrón. Lo que ocurre es que Python busca subcadenas que correspondan, no busca la correspondencia de toda la cadena pasada.

Por el momento hemos usado el método *match()* que solo busca la correspondencia al principio de la cadena o en la posición que le indiquemos (en nuestro caso la posición *1*).

```
>>> print p.match("cabaaab"➥
,0)
None
>>> print p.match("abaaab",0)
<_sre.SRE_Match object at➥
0x81c4c60>
>>>
```

Existe otro método llamado *search()* que se comporta de manera parecida a las expresiones regulares de Perl. Busca todas las correspondencias y es posible obtener una lista de ellas. Pero lo más útil es el método *findall()*, devuelve una lista con todas las apariciones del patrón.

```
>>> resultado = p.search(➥
"abaaabcaaaa",1)
```

```
<_sre.SRE_Match object at
0x81b4800>
>>>
>>> p.findall("abaaabcaaa",0)
['abaaab', 'aaa']
```

Introducción a Gnuplot

Gnuplot (véase recurso [1]) es un programa de dibujado de datos. Se le pasan datos o una función matemática y se usa la orden *plot* para hacer la gráfica. Gnuplot entonces abre una ventana y dibuja esos datos usando los parámetros que le especifiquemos.

Cuando ejecutamos Gnuplot nos aparecerá un shell. Gnuplot tiene sus propios comandos y lenguaje. Lo mejor de tener un shell es que podemos jugar con él, vamos a hacerlo:

```
gnuplot> f(x) = sin(x)
gnuplot> plot f(x)
gnuplot>
```

Veremos que aparece una ventana como la mostrada en la Figura 1. Es la función seno, que hemos dibujado con solo dos instrucciones. Parece interesante ¿no?. La orden *plot* acepta muchos parámetros. Podemos dibujar muchas funciones a la vez, usar pequeños cuadraditos o puntos para diferenciarlas, interpolar puntos, etc. Gnuplot permite además realizar muchos tipos de gráficas: de dos dimensiones, de tres dimensiones, histogramas…

Una característica que nos va a resultar de mucha utilidad es que permite exportar las gráficas a formatos gráficos estándar, como gif o png. Podemos hacer todas la pruebas usando como salida una ventana de Xwindows y cuando estemos seguros de los comandos y parámetros necesarios usar como salida un archivo gráfico.

El programa Gnuplot

Gnuplot está diseñado para que nos permite automatizar su uso a través de scripts compuestos por sus propios comandos. La idea es poder usarlo en una de esas famosas tuberías a través de las que los usuarios de UNIX más avanzados pueden crear aplicaciones enteras.

Gnuplot necesita un *terminal* para funcionar que actúa como driver de impresión. Este *terminal* puede ser o bien una máquina, como una impresora láser, o un fichero gráfico, como un fichero JPEG, o directamente una ventana gráfica en Xwindow. Aprovechando esta característica podemos desarrollar la configuración de una gráfica directamente en el shell de Gnuplot visualizando en una ventana. Una vez que sepamos la configuración necesaria, solo tenemos que cambiar el *terminal*.

Las variables se pueden ver usando el comando *show*:

```
gnuplot> show terminal
   terminal type is x11
gnuplot>
```

Listado 1:gnuplotter.py

```
01 #!/usr/local/bin/python
02
03 from Numeric import *
04 import Gnuplot,
   Gnuplot.funcutils
05 import re;
06
07 class Grafica:
08
09  def __init__(self, cadena):
10       self.g     =
   Gnuplot.Gnuplot(debug=1)
11   self.g.title('Mi gráfica')
12    self.g('set data style
   linespoints')
13    self.g('set boxwidth 0.9
   absolute')
14   self.g('set style fill solid
   1.000000 border -1')
15   self.g('set style histogram
   clustered gap 5 title 0, 0')
16    self.g('set style data
   histograms')
17   self.g('set title \"Lineas
   con ' + cadena +' \"')
18
19   self.hash = {}
20   self.cadena = cadena
21
22  def cargaDatos(self,
   fichero):
23
24   archivo = file(fichero,'r')
25
26   self.hash = {}
27
28         expr       =
   re.compile(self.cadena);
29
30   for linea in archivo:
31     if( re.search(expr,
   linea)):
32     lista = linea.split(' ')
33       fecha = lista[0]+'
   '+lista[1]
34            if    ( not
   self.hash.has_key(fecha)):
35      self.hash[fecha] = 1
36     else:
37        self.hash[fecha] =
   self.hash[fecha] + 1
38
39   archivo.close()
40
41
42  def dibuja(self):
43
44   llaves = self.hash.keys()
45   llaves.sort()
46
47
48   etiquetas = "set xtics ("
49
50   i = 0
51   lista = []
52
53   for llave in llaves:
54    etiquetas = etiquetas +
   "\"" + llave + "\" " + str(i)
   + " ,"
55
   lista.append([self.hash[llave]
   ,i])
56    i = i+1
57
58   etiquetas = etiquetas +
   "\"fin\"" + str(i) + ")"
59
60
61    self.g('set xrange
   [0:'+str(i-1)+']')
62   self.g('set yrange [0:30]')
63
64   self.g(etiquetas)
65
66   self.g.plot(lista)
67    raw_input('Pulsa
   INTRO...\n')
68   self.g.reset()
69
70
71 if __name__ == "__main__":
72  g = Grafica('ssh')
73  g.cargaDatos('prueba.txt')
74  g.dibuja()
```

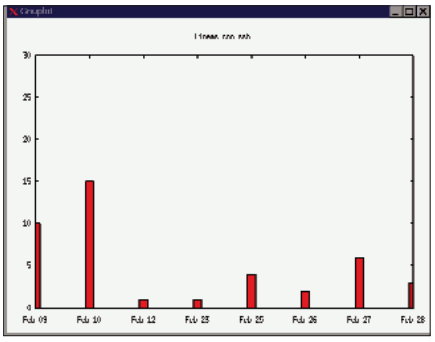

Figura 2: Utilizamos gnuplot para mostrar el número de conexiones ssh de manera gráfica.

Usando el comando *help* podemos ver los distintos terminales soportados, que son unos cuantos. Por el momento solo nos van a interesar los terminales *x11* y JPEG. Para cambiar el tipo de terminal tenemos que cambiar el valor de la variable. Para eso está el comando *set*:

```
gnuplot> set terminal pstricks
Terminal type set to 'pstricks'
gnuplot> set terminal x11
Terminal type set to 'x11'
Options are '0'
gnuplot>
```

Una vez definido el *terminal* hemos de configurar los parámetros de la gráfica. En Gnuplot se pueden configurar muchos parámetros que afectan a la gráfica. Por ejemplo, si no especificamos nada y solo le damos una serie de puntos los pondrá en la gráfica, pero no los unirá con una linea. También acepta parámetros sobre la precisión de los intervalos que usará en los ejes.

Las variables se configuran con el comando *set*. Usando la orden de ayuda *help set* podemos ver todas las variables que podemos modificar. Por ejemplo, para poner título a la gráfica se usa:

```
gnuplot> set title➥
"Nuestra gráfica"
```

Cuando las variables han sido configuradas podemos realizar el dibujado mediante el comando *plot*. *plot* tiene muchas opciones y permite dibujar varias gráficas juntas. Cada gráfica va acompañada de los parámetros que la definen, por ejemplo:

```
gnuplot> plot sin(x)➥
using impulses
```

Dibujará la función seno usando lineas desde el eje en lugar de una linea que una los puntos.

Gnuplot y Python

Para poder utilizar Gnuplot en Python necesitamos descargar e instalar las librerías py-gnuplot (véase recurso[2]) o mediante un paquete de nuestra distribución de Linux. Obviamente, también será necesario instalar el programa Gnuplot. En particular usaremos un método anónimo que nos permite mandar comandos a Gnuplot:

```
>>> g('set title \➥
"Nuestra Gráfica\"')
```

Una vez que tengamos el entorno configurado, invocaremos el método "plot()". La librería "py-gnuplot" se ha diseñado para ahorrarnos trabajo en el uso de Gnuplot y permitir el uso de las funciones de "Numeric" como fuente de datos. Podemos realizar gráficas a partir de:

- Ficheros
- Listas de Python
- Funciones de Numeric

Nosotros dibujaremos a partir de una lista que generamos en nuestra búsqueda con expresiones regulares. Una peculiaridad de py-gnuplot es que realiza el dibujado de la gráfica muy rápido. Si invocamos a *plot()* sin más ¡no veremos nada! Para parar el programa cuando se realiza el dibujado llamaremos a *raw_input()* justo entre *plot()* y *reset()*.

```
>>> a = [[1,2], [2,2], [4,5]]
>>> g.plot(a)
>>>
```

Dibujará la gráfica que formen los puntos (1,2), (2,2) y (4,5).

Nuestro programa

Veamos ya qué hace nuestro programa, cuyo código podemos ver en el Listado 1. Tenemos una clase llamada *Gráfica* con 3 métodos. El primero es *__init__()* y lo que hace es preparar el entorno de Gnuplot. Asigna ciertas variables y declara una variable que llamamos "hash". Además guarda en una variable de instancia la cadena que usaremos como patrón.

El segundo método es *cargaDatos()*. Se encarga de abrir un fichero en modo de lectura, compilar la expresión regular y, para cada línea del fichero, comprobar si esa línea contiene la expresión regular que buscamos. En caso de que así sea dividimos la linea usando como separador el espacio(" "). Se cogen las dos primeras palabras de la linea, que formarán una fecha, y se usan como llave para un diccionario que hemos llamado "hash". Si esa fecha no estaba ya en el diccionario, la introducimos con un valor de 1. Si ya estaba incrementamos su valor en 1. De esta manera llevamos la cuenta del número de apariciones del patrón por fecha. El resultado final será un diccionario que almacenarán las ocurrencias del patrón por fecha.

El tercer método es *dibuja()*. Asigna etiquetas a los puntos del eje x de la gráfica a través de la variable "xtics". Esas etiquetas serán las fechas que usamos como llaves en el diccionario *hash*. Además vamos preparando una lista que contiene las coordenadas X e Y de los puntos a dibujar. Se asignan rangos para el eje X e Y y posteriormente se dibuja.

El programa se compone de la creación del objeto *Grafica*, la invocación de su método *cargaDatos()* con el nombre de un fichero y la invocación del método *dibuja()*. Como resultado final tendremos una gráfica parecida a la mostrada en la Figurado 2. ■

RECURSOS

[1] Gnuplot: *http://www.gnuplot.info*

[2] Py-Gnuplot: *http://gnuplot-py.sourceforge.net*

[3] Regular Expressions HOWTO: *http://www.amk.ca/python/howto/regex/*

[4] Numeric: *http://http://numeric.scipy.org/*

LOS AUTORES

José María Ruiz actualmente está realizando el Proyecto Fin de Carrera de Ingeniería Técnica en Informática de Sistemas. Lleva 7 años usando y desarrollando software libre y, desde hace dos, se está especializando en FreeBSD, actualmente trabaja en Animatika una empresa malagueña dedicada al software libre. Pedro Orantes está cursando 3º de Ingeniería Técnica en Informática de Sistemas en Madrid, y está desarrollando su Proyecto Fin de Carrera con Jython.