

El uso de expresiones regulares en Python

# REGULARIDAD

Todo administrador de sistemas o programador debe enfrentarse tarde o temprano a las expresiones regulares. Quizá sean una de las herramientas que más pueden ayudarnos en nuestro trabajo diario. Vamos a crear un programa con Ironpython y Gtk# que nos permitirá ver cómo actúan antes de emplearlas. **POR JOSÉ MARÍA RUIZ**

**M**e encantan, las expresiones regulares son una de esas cosas que aprendes una vez y siempre usarás. Hay un antes y un después. Las pudimos ver en un número anterior, por lo que no vamos a repasarlas aquí. Pero hay algo que hace que no todo el mundo se atreva a usarlas: lo difícil que es verlas en acción.

Da igual cuanto hayas usado las expresiones regulares, siempre tienes que hacer un par de pruebas antes de estar seguro de que funciona perfectamente esa expresión que has estado preparando. Así que te dedicas a ejecutar prueba tras prueba buscando afinarla. Entrás en el editor, escribes la expresión dentro de una función, pasas al shell, ejecutas el script Python, ves el resultado (que estará mal), y vuelves al editor de nuevo en un ciclo que sólo acaba cuando la expresión se comporta como debería hacerlo desde el principio.

En este artículo vamos a emplear IronPython y Gtk#, la versión Mono de Gtk, para

desarrollar un programa simple pero muy útil: un afinador gráfico de expresiones regulares.

## ¿Por qué Ironpython y Gtk#?

¿Y por qué no? Ironpython, ver Recurso [1], es un intérprete de Python creado sobre .Net o Mono (ver Recurso [2]) en el mundo del software libre. Python es muy elegante, muy simple... y muy lento. Reconozcámoslo, tanto Python como Ruby adolecen de un gran problema de rendimiento. Ironpython emplea toda la investigación desarrollada tanto en .NET como en Mono sobre optimización. El resultado es que Ironpython es dos veces más rápido que el intérprete de Python estándar.

¿Y no tiene algún tipo de problema? Sólo uno, no cuenta con parte de la librería de Python debido a que determinadas funciones están programadas en C y son específicas de cada sistema operativo. Pero este problema se compensa con la posibilidad de

reutilizar código Python así como de acceder a todas las librerías de Mono y .NET.

Y de este hecho proviene otra ventaja en cuanto a desarrollo: Gtk#. Gtk# es una librería que da acceso a todo el framework Gtk desde .NET. Simplifica enormemente el desarrollo de interfaces, tanto que estoy tentado a decir que es tan simple de desarrollar un GUI en Gtk# e Ironpython como con Tcl/Tk (y quien conozca Tcl/Tk sabe a qué me refiero).

Armados con estas dos formidables herramientas vamos a desarrollar una aplicación que nos facilitará la vida cuando desarrollemos expresiones regulares en nuestros programas Python, y de paso veremos cómo se comporta Ironpython con Gtk#.

## Ironpython

Instalar Ironpython consiste básicamente en instalar Mono en nuestra distribución, si no se encuentra ya instalado por defecto, e instalar Ironpython. Un vez compilado, Ironpython se comporta como Python, sólo que su intérprete de comandos es un desastre. Sí, no me gusta nada, no tiene historial y para salir de él, como ya vimos en un artículo

### Listado 1: Ejemplo

```

01 import clr
02
03 clr.AddReference('gtk-sharp')
04 import Gtk
05 def hola (objeto,
06           argumentos):
07     print "Hola a todos los
08         linuxeros"
09
10 def cerrar_ventana (o,
11                   args):
12
13     Gtk.Application.Quit ()
14
15     Gtk.Application.Init ()
16
17     w = Gtk.Window
18         ("Comprobador de regexp")
19     w.DefaultWidth=100
20     w.DefaultHeight=50
21     w.DeleteEvent +=
22         cerrar_ventana
23     caja = Gtk.VBox(False,4)
24     boton = Gtk.Button
25         ("Comprobar")
26     boton.Clicked += hola
27     caja.Add(boton)
28     w.Add(caja)
29     w.ShowAll ()
30     Gtk.Application.Run ()

```

anterior, casi hay que hacer magia negra (presionar Control-Z e intro).

Por tanto, es mejor desarrollar usando un editor de textos y posteriormente ejecutar usando el comando:

```
$ ironpython -c 2+2
4
```

Si ejecutas el comando que aparece arriba desde un shell, te darás cuenta de que Ironpython es lento arrancando. Se debe a que ha de arrancar la máquina virtual de Mono, muy parecida a la de Java. En ambos casos, es durante el arranque cuando notaremos el problema de lentitud debido a que una vez en funcionamiento son muy rápidas.

Muy bien, ya tenemos Ironpython, ahora necesitamos Gtk#. Para ello sólo debemos instalar esta librería en nuestro sistema. Generalmente no la encontraremos como Gtk#, sino como *gtk-sharp*, puesto que «#» se lee en inglés «sharp». Una vez instalada sólo debemos probar a cargarla:

```
import clr
clr.AddReference('gtk-sharp')
import Gtk
```

Primero debemos importar la librería *clr* que nos da acceso a todas las librerías .NET de Mono, y posteriormente hay que añadir una referencia a la librería *gtk-sharp* que no forma parte de las librerías base de Mono. Al añadir esta referencia hacemos que *clr* busque y cargue la librería .NET de Gtk#. Con todas las piezas en su sitio importamos *Gtk*. Este hecho es algo maravilloso, porque Gtk# no posee ningún enlace o librería puramente Python, .NET nos permite cargar librerías que han sido desarrolladas en otros lenguajes .NET, aunque no sea en nuestro lenguaje de forma transparente. Es absolutamente dinámico, nadie ha tenido que desarrollar un enlace Python para que Ironpython pueda emplear esa librería. Los nombres de las funciones se traducen automáticamente, así como los tipos de datos.

Ya tenemos Ironpython y Gtk#, vayamos a por nuestro programa.

## Creamos un GUI

Un programa Gtk consiste en una serie de elementos gráficos, o *widgets*, que posicionamos dentro de una ventana, más una serie de funciones que son ejecutadas en base a eventos provocados por esos *widgets*. Por tanto, debemos crear los *widgets*, posi-

cionarlos, y vincular eventos, widgets y funciones.

Una vez hecho esto pasamos el control del programa al bucle de gestión de eventos de Gtk, que se dedicará a identificar eventos y ejecutar las funciones asociadas. Un programa Gtk tendrá básicamente el siguiente aspecto:

```
Gtk.Application.➤
Init ()
# código
# código
# más código
Gtk.Application.Run ()
```

Todo el programa debe existir entre estas dos funciones como si fuese un sandwich. La primera, *Gtk.Application.Init()* arranca Gtk, procesando los parámetros de línea de comandos si los hubiese e inicializando estructuras de datos. A continuación, en mitad del sandwich, viene nuestro código. Por último ejecutamos la función *Gtk.Application.Run()*: pasa el control al bucle de gestión de eventos de Gtk que no parará hasta que el programa pare, por las buenas o por las malas.

¿Qué tenemos que meter en mitad de este sandwich? La verdad es que el proceso es bastante repetitivo: crear *widget*, configurar *widget* y posicionar *widget*. Con :

```
boton = Gtk.Button("Prueba➤
GTK#")
```

Creamos un botón Gtk. Crear un *widget* no tiene ningún resultado visual en nuestro programa. Podemos crear un millón de *widgets* y que nuestra ventana siga pareciendo un árido desierto. Un *widget* es un elemento bastante independiente, por lo que hay que configurarlo si fuese necesario para que se muestre algo más sociable... y útil:

```
def hola (objeto,➤
argumentos):
    print "Hola a todos los➤
linuxeros"
boton.Clicked += hola
```

Con este código hacemos dos cosas. La primera es bastante tonta, creamos una función que al ejecutarse simplemente muestra por el terminal «Hola a todos los linuxeros», lo interesante está en el segundo paso.

.NET aplica muchas técnicas que se han ido descubriendo durante los años en el mundo de la programación orientada a

objetos. Una de ellas es la creación de grupos de *Listeners*, que podemos traducir como «oyentes». *b.Clicked* es un evento, una noticia que se dispara cuando el botón *b* es pulsado (*Clicked* en inglés). .NET nos permite hacer algo bastante curioso. Cada evento es una especie de cola a la que podemos añadir funciones. Estas funciones, u oyentes, permanecen ahí como si estuviesen escuchando, agazapadas, a que se de el evento en cuestión, momento en el cual se ejecutarán automáticamente.

El empleo del operador `+=`, que generalmente es matemático, simplifica la sintaxis y viene a ser como una especie de *append()*, que se añade al final de la cola de *listeners*.

Por tanto, ahora nuestro botón ya reacciona cuando es pulsado, el problema es que para pulsarlo hay que verlo primero y aún permanece oculto.

Pero, claro, tenemos un problema. Si hacemos que aparezca... ¿dónde aparecerá? Hay que posicionar el *widget* en la ventana. Hablando de la ventana, estamos constantemente haciendo referencia a los botones, pero ella debe aparecer por algún sitio.

```
w = Gtk.Window➤
("Comprobador de regexp")
w.DefaultWidth=100
w.DefaultHeight=50
w.DeleteEvent += cerrar_ventana
```

¡Ya tenemos ventana! No sólo eso, la hemos configurado para que tenga un tamaño de 100 píxeles de ancho por 50 píxeles de alto. La propia ventana es un *widget* de Gtk#, por lo que la creamos como lo hicimos con el botón. Tanto la altura como el ancho los asignamos a lo que parecen dos variables, pero que en .NET se denominan *properties* o propiedades del objeto. Básicamente son funciones que parecen variables. De esta forma, cuando asignamos una propiedad .NET se pueden estar realizando una serie de trabajos detrás de las cortinas. En este caso, estos trabajos son la preparación de una ventana de un tamaño determinado. De nuevo vemos cómo asignamos una función, *cerrar\_ventana*, a un evento de igual forma a como hicimos con el botón.

Vale, tenemos una ventana y un botón, pero seguimos sin ver nada y seguimos sin posicionar el botón. Para esta tarea existen unos *widgets* especiales cuya labor es la de contener otros *widgets* y posicionarlos. En Gtk se denominan cajas, *boxes*, y las tenemos de dos tipos: verticales y horizontales.

## Listado 2: Programa Regexp

```

001 import clr
002
003 import Gtk
004 import re
005
006 class programa :
007     def __init__ (self):
008         Gtk.Application.Init
009         ()
010         self.ips = []
011         self.lista = []
012         self.ip_label = []
013         self.ip_barra = []
014         self.genera_gui()
015         Gtk.Application.Run ()
016
017     def genera_gui (self):
018         self.w = Gtk.Window
019         ("Comprobador de regexp")
020         self.w.DefaultWidth=800
021         self.w.DefaultHeight=600
022         self.w.DeleteEvent +=
023         self.cerrar_ventana
024         self.box =
025         Gtk.VBox(False,4)
026
027         # El menú
028         self.mb = Gtk.MenuBar
029         ()
030         self.file_menu =
031         Gtk.Menu ()
032
033         self.imprime_item =
034         Gtk.MenuItem("Imprime")
035         self.file_menu.Append
036         (self.imprime_item)
037
038         self.exit_item =
039         Gtk.MenuItem("Salir")
040         self.file_menu.Append
041         (self.exit_item)
042
043         self.file_item =
044         Gtk.MenuItem("Archivo")
045         self.file_item.Submenu
046         = self.file_menu
047
048         self.mb.Append
049         (self.file_item)
050
051         self.box.PackStart(self.mb
052         , False, False, 0)
053
054         # La caja de texto
055         self.caja_texto =
056         Gtk.VBox(False,4)
057         self.texto =
058         Gtk.TextView()
059         cadena =
060         self.texto.Buffer
061         cadena.Text = "Hola
062         mundo"
063
064         # La caja con los
065         widgets
066         self.box2 = Gtk.VPaned
067         ()
068
069         self.caja_form =
070         Gtk.HBox(False,4)
071         self.b = Gtk.Button
072         ("Comprobar")
073         self.e = Gtk.Entry ()
074
075         # Empaquetamos los
076         widgets
077         self.caja_texto.Add(self.t
078         exto)
079
080         self.box.Add(self.box2)
081
082         self.box2.Add1(self.caja_f
083         orm)
084
085         self.box2.Add2(self.caja_t
086         exto)
087
088         self.caja_form.Add(self.e)
089
090         self.caja_form.Add(self.b)
091         self.w.Add (self.box)
092
093         self.w.ShowAll ()
094
095         # Poner un TextView
096         # Conectamos las
097         señales
098         self.b.Clicked +=
099         self.marca_regex
100
101         self.exit_item.Activated
102         += self.cerrar_ventana
103
104         self.imprime_item.Activate
105         d += self.imprime
106
107         tabla =
108         self.texto.Buffer.TagTable
109
110         tag =
111         Gtk.TextTag("regexp")
112         tag.Background = "red"
113         tag.Foreground =
114         "white"
115         tabla.Add(tag)
116
117         tag =
118         Gtk.TextTag("subregexp")
119         tag.Background =
120         "blue"
121         tag.Foreground =
122         "white"
123         tabla.Add(tag)
124
125         def marca_regex (self,
126         o, args):
127             texto =
128             self.texto.Buffer
129             texto.RemoveAllTags(texto.
130             StartIter,texto.EndIter)
131             texto =
132             self.texto.Buffer
133             mi_regexp =
134             self.e.Text
135             try:
136                 iter =
137                 re.finditer(mi_regexp,text
138                 o.Text)
139                 for ma in iter:
140                     def colorea
141                     (inicio,fin,tipo) :
142                         iter_inicio
143                         =texto.GetIterAtOffset(ini
144                         cio)
145                         iter_fin
146                         =texto.GetIterAtOffset(fin
147                         )
148                         texto.ApplyTag(texto.TagTa
149                         ble.Lookup(tipo),iter_inic
150                         io,iter_fin)
151                         inicio,fin =
152                         ma.span()
153                         colorea(inicio,fin,"regexp
154                         ")
155                         # ¿Hay más grupos?
156                         if (ma.lastindex
157                         != None):
158                             for i in
159                             range(0,ma.lastindex):
160                                 print "Grupo:
161                                 %d, %s" % (i+1,
162                                 ma.group(i))
163                                 inicio,fin =
164                                 ma.span(i+1)
165                                 colorea(inicio,fin,"subreg
166                                 exp")
167             except Exception:
168                 m =
169                 Gtk.MessageDialog(self.w,G
170                 tk.DialogFlags.DestroyWith
171                 Parent,Gtk.MessageType.Inf
172                 o,Gtk.ButtonsType.None,"Ex
173                 presion Regular con
174                 errores")
175                 m.Show()
176
177             def cerrar_ventana
178             (self, o, args):
179                 Gtk.Application.Quit
180                 ()
181
182             def imprime (self, o,
183             args):
184                 print self.e.Text
185
186             p = programa()

```

Son los sistemas de posicionamiento más simples, los hay mucho más complicados, pero nos valen para lo que queremos hacer. Se crean exactamente igual a como hemos creado los anteriores *widgets*:

```
caja = Gtk.VBox(False,4)
```

La *V* viene de la palabra vertical. Los argumentos son simples: el primero le dice a la caja que no guarde los *widgets* de forma homogénea, todos del mismo tamaño, mientras que el segundo indica que queremos que hayan 4 píxeles de holgura alrededor de los *widgets* contenidos.

Ya tenemos todos los ingredientes listos para posicionar nuestro botón:

```
caja.Add(boton)
w.Add(caja)
```

Y sólo nos queda el toque final:

```
w.ShowAll ()
```

Esta función de la ventana hará visibles todos los *widgets* que existan en su interior. Ya tenemos nuestro primer programa *Gtk#* (ver Listado [1])

## Los widget TextView y TextBuffer

Aunque parezca increíble, con lo visto en el apartado anterior es posible comprender todo el código del Listado [2]. El único problema puede aparecer con los *widgets*, *TextView* y *TextBuffer*. *TextWidget* se encarga de la parte visual de la representación de texto, y lo cierto es que trabajaremos poco con él. Es *TextBuffer* quien hace el trabajo duro, puesto que se encarga de gestionar el texto de *TextView*. Podemos acceder al *TextBuffer* de la siguiente manera:

```
textview = Gtk.TextView()
textbuffer = textview.Buffer
textbuffer.Text = "Hola a todos"
```

Como podemos observar, *Gtk#* es bastante sencillo, creamos el *TextView*, extraemos el *TextBuffer* y cargamos en él algo de texto mediante una propiedad. Pero queremos algo más, queremos colorear texto. En nuestro programa permitiremos al usuario escribir una expresión regular, y un *TextView* con texto coloreará las porciones del mismo que se correspondan con la expresión regular. ¡Es algo realmente útil! Con un par de pruebas el lector quedará convencido.

¿Cómo podemos colorear texto? Para ello emplearemos otros objetos de *Gtk* llamados *TextTags*, «etiquetas» en castellano. Estos *tags* son configuraciones de fuentes de texto y colores que se asocian a un nombre. Por ejemplo, podemos hacer un *tag* para que en el *TextView* todas las apariciones de la palabra «Linux Magazine» sean en negrita. Pero los *tags* no se ocupan de esto. Sólo sirven para asociar propiedades del texto a una cadena de texto ¿Con qué objetivo?

*TextBuffer* posee una propiedad que contiene una tabla interna que nos permite almacenar tantos *TextTags* como queramos. Una vez almacenados podemos hacer que a ciertas zonas de texto, delimitadas por dos posiciones, se les asocie un *tag*, y que por tanto cambien su aspecto. Asignamos los colores de fondo, background, y primer plano, foreground, mediante propiedades del *tag*:

```
tabla = textbuffer.TagTable
tag = Gtk.TextTag("colorines")
tag.Background = "red"
tag.Foreground = "white"
tabla.Add(tag)
```

Las posiciones dentro de un *TextBuffer* se especifican mediante un iterador, y aunque suene muy complicado, en realidad es sencillo:

```
iter = textbuffer.➤
GetIterAtOffset(0)
```

Con esta simple llamada conseguimos un iterador posicionado en el carácter de la posición 0 del *TextBuffer*, y con un par de iteradores y un *tag*...

```
textbuffer.ApplyTag(texto.➤
TagTable.Lookup("colorines"), ➤
iter_inicio,iter_fin)
```

Y ya tenemos una región del texto con los colores que hemos asociado al *tag* «colorines». También tenemos casi todo lo que necesitamos para que nuestro programa funcione. Sólo falta el motor del mismo, el comprobador de expresiones regulares

## Trasteando con Expresiones Regulares

Tenemos una caja de texto en la que el usuario introducirá una expresión regular (o quien sabe qué cadena de texto) y que al pulsar el botón comprobar debería iluminar como un árbol de navidad el texto que aparece en el *TextView* que se encuentra debajo.

Para controlar los errores generados por una expresión regular defectuosa recurriremos al control de excepciones, introduciendo todo el código que trastea con la expresión regular dentro de un gran *try...catch*. La ventana que mostrará el mensaje será un *MessageDialog* que ya viene preconfigurado con diversos esquemas de mensaje, en nuestro caso lo configuramos como de tipo *Info* (al final no hemos sido tan duros con el pobre usuario... lo mismo es porque seremos nosotros mismos, es una de las ventajas de ser el programador).

Distingo dos tipos de aciertos. Las expresiones regulares pueden contener grupos, partes de la expresión enmarcados por dos paréntesis. Estos grupos son devueltos por el motor de expresiones regulares junto con toda la expresión en sí, y suelen ser bastante útiles. Usaremos un *tag* diferente para toda la expresión regular y los distintos grupos, fondos rojo y azul respectivamente como se muestra en el Figura 1.

El procedimiento consiste en sacar el texto del *TextView*, pasarle la expresión regular con *finditer()*, que va recorriendo el texto y nos permite iterar sobre los aciertos, y para cada acierto colorear tanto expresión como los grupos si los hubiese. La función *span()* nos devuelve las posiciones de inicio y fin de cada grupo de una expresión regular en el texto sobre el que la ejecutamos. Usando esas dos posiciones para obtener los iteradores de *TextBuffer* podremos aplicar los *tags*. El grupo 0 es siempre el texto que se corresponda con toda la expresión regular, por lo que lo trataremos por separado debido a que lo pintaremos de rojo.

Y poco más se puede decir del programa, salvo que es muy útil (el autor ha ahorrado muchas horas de prueba y errores con programas similares).

## Conclusión

Ironpython y *Gtk#* son una opción nada desdenable para la construcción de programas gráficos simples y eficientes. Es difícil encontrar una librería que permita crear un interfaz gráfico de forma tan sencilla y con código tan legible, lo que posibilita que sea cada más sencillo crear pequeños programas gráficos con los que resolver tareas tediosas del día a día. ■

## RECURSOS

[1] IronPython: <http://www.codeplex.com/IronPython>

[2] Mono: <http://www.mono-project.com>