

Controlar tu información desde un solo punto con Python

MASH-UPS

Twitter-Facebook-Rss, GoogleMaps-Noticias-Mensajería,... parece que los mash-ups tienen como objetivo distraernos. Creemos uno que nos ayude. **POR JOSÉ MARÍA RUÍZ**

Nadie quiere estar consultando constantemente distintas webs y programas. Es tanta la información que nos llega hoy día, que la gente está inventando todo tipo de filtros para quedarse con lo importante. Existen incluso editores de texto que bloquean el acceso a Internet para ayudar a la concentración del que escribe. Yo no sé qué pensará el lector, pero hay días en los que desearía que Internet se apagase sola un rato.

Como no podemos permitirnos ese lujo – me imagino el sonoro grito de angustia de centenares de millones de almas – lo que sí podemos es hacer trabajar a nuestro ordenador por nosotros y que nos filtre parte del contenido. Se supone que los ordenadores están para ayudarnos a tener más tiempo, o ¿no era así como los vendían hace 20 años?

Como esta pretensión es muy ambiciosa, creemos algo sencillo y práctico que nos permita adaptarlos a nuestras necesidades personales y ampliarlo a nuestro antojo (es posible ver el código en el Listado 1).

Vamos a buscar la máxima simplicidad en nuestro proyecto. Las distintas fuentes de información a las que nos conectaremos deben ser consultadas cada cierto

tiempo. Digamos que queremos saber si hay novedades en Twitter, en tal caso conectaríamos con Twitter cada cinco minutos y descargaríamos los mensajes dejados por las personas a las que seguimos. Es vital que nuestro sistema nos permita hacer *pooling*, ejecutar tareas cada cierta cantidad de tiempo.

Además, queremos poder acceder en cualquier momento a nuestra aplicación. La solución a este problema es emplear una web. Podemos configurar el router de nuestra casa para que nos permita consultar nuestros datos desde cualquier sitio y en cualquier momento. Por ello, vamos a montar el sistema sobre un servidor web.

Analizaremos este programa partiendo de la recolección de los datos y acabando en el interfaz.

IMAP

IMAP se ha convertido en uno de los protocolos de acceso a correo electrónico más importantes del mundo. Es rara la cuenta de correo electrónico que no permite la consulta mediante IMAP, y Gmail lo permite casi desde el principio.

Python dispone desde hace bastante tiempo de una librería para la gestión de correo con IMAP llamada *imaplib*

[2]. Como es parte de la librería base de Python, dispondremos de ella allá donde tengamos Python instalado. Siempre es aconsejable hacer uso de la librería base de Python cuando sea posible, puesto que así conseguiremos que nuestro software no dependa de librerías externas que no controlamos y que requieren instalación, como nos pasará más adelante.

El protocolo IMAP nos permite trabajar con los correos electrónicos directamente en el servidor, sin necesidad de descargarlos. Para ello, IMAP establece una serie de protocolos que, siendo sinceros, pueden ser bastante liosos. Pero antes de poder trabajar con los correos necesitamos una conexión con el servidor de correo:

```
>>> from imaplib import IMAP
>>>
>>> con = IMAP4(
("mi.servidor.com")
>>> print con.login("usuario",
"miclave")
('OK', ['Logged in.'])
```

Con estos sencillos pasos obtenemos una conexión con nuestro servidor IMAP. Si nuestro servidor emplea

cifrado SSL para asegurar la transmisión de los datos, deberemos utilizar la clase *IMAP_SSL* en lugar de *IMAP*.

Un servidor IMAP se compone de carpetas, éstas de mensajes y los mensajes de partes. Esta división en forma de árbol es vital para comprender los siguientes pasos, el primero de los cuales consiste en seleccionar la carpeta en la que queremos trabajar:

```
>>> con.select('INBOX')
```

INBOX es una carpeta que debe aparecer en toda cuenta IMAP, es obligatoria. Es posible que dispongamos de muchas otras carpetas, opcionales, y si queremos controlar los correos de una de ellas, deberemos seleccionarla como hemos hecho con *INBOX*. Una vez dentro de nuestra carpeta, vamos a buscar todos los correos que se correspondan con un patrón específico:

```
>>> status, lista_mensajes = \
con.search(None, "FROM", \
"linux-magazine")
```

Como estoy ya fuera de plazo para entregar mi artículo mensual a Linux Magazine, quiero controlar cualquier correo que venga de esa dirección. Si quisiéramos todos los correos de la carpeta, deberíamos haber pasado a *search()* como segundo parámetro la cadena "ALL". La búsqueda es aproximada, así "linux-magazine" vale tanto para la dirección *imprensa@linux-magazine.es* como para *director@linux-magazine.com*.

search devuelve como resultado dos valores. El primero sirve para indicarnos si la consulta ha tenido éxito o no, mientras que el segundo es el resultado propiamente dicho, y es el que nos interesa. Se compone de una tupla, donde el segundo valor es una cadena que contiene los números de identificación de los correos que coinciden con lo buscado. En IMAP cada correo dispone de un identificador único, y para obtenerlo podemos usar:

```
>>> for id_mensaje in \
lista_mensajes[0].split():
>>> cmd = "(BODY \
[HEADER.FIELDS \
(SUBJECT DATE FROM))"
>>> respuesta, \
```

```
mensaje = \
con.fetch(id_mensaje, \
cmd)
>>> msg = \
email.FeedParser.\
FeedParser()
>>> msg.feed \
(mensaje[0][1])
>>> the_email = \
msg.close()
```

Es aquí donde IMAP cobra sentido. Cada mensaje se compone de partes, pudiendo solicitar al servidor IMAP que sólo nos devuelva algunas de ellas. En este caso hemos pedido que nos devuelva los campos *SUBJECT* (asunto), *DATE* (fecha) y *FROM* del correo. No queremos ni el texto, ni los ficheros que pudiesen ir adjuntos al correo. De esta forma podemos movernos rápidamente entre los correos, aunque algunos «pesen» unos pocos megas.

imaplib nos devuelve estos campos como líneas de texto, y la verdad es que es bastante complicado procesarlos.

¡No hay problema! La librería base de Python dispone de una potente herramienta, la librería *email*. Esta librería dispone de un procesador de correos llamado *FeedParser* que puede procesarlos poco a poco, no es necesario tener un correo completo para que funcione. En el código anterior empleamos este procesador, pasándolo al texto del correo (que está en *mensaje[0][1]*). Antes de poder obtener nuestro correo procesado hay que indicarle a *FeedParser* que no le daremos más datos mediante el método *close()*, y como resultado obtendremos un diccionario como el siguiente:

```
{"From" : "josemaria @miweb.
es",
"Subject" : "Hola a todos",
"Date" : "Thu, 25 Feb \
2010 22:15:37 +0100"}
```

Almacenando este tipo de datos en una lista, dispondremos de los que necesitamos de nuestra cuenta de correo para mostrarlos más adelante.

Twitter

El asunto con Twitter [3] es mucho más sencillo que con IMAP. Emplearemos la

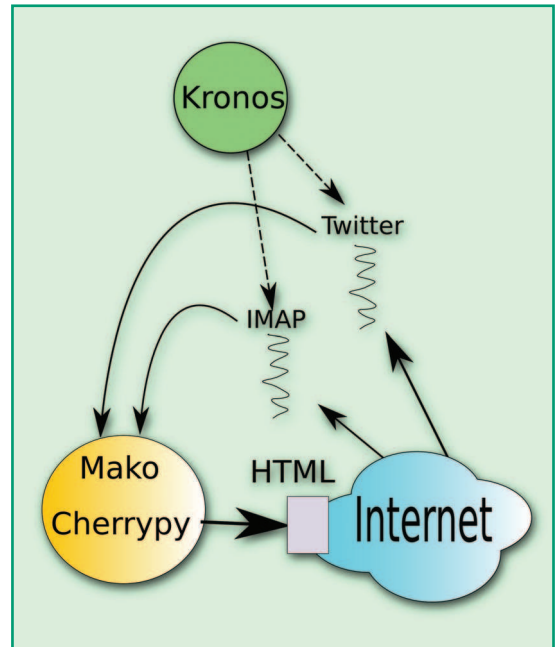


Figura 1: Esquema de funcionamiento.

librería *python-twitter* [4], que expone el api de Twitter en Python de forma sencilla. Vamos a obtener de Twitter los últimos tweets de unos cuantos twitters (¿existe este nombre?) en los que estamos interesados:

```
01 >>> import twitter
02 >>>
03 >>> api = twitter.Api()
04 >>> lista =
api.GetUserTimeline
(u"linux_spain")
05 >>> tweets = []
06 >>> for elem in lista:
07 >>> dic = tweet.asAdict()
08 >>> segundos =
tweet.GetCreatedAtInSeconds()
09 >>> fecha =
datetime.fromtimestamp
(segundos)
10 >>>
tweets.append ({ "nombre"
: dic["user"]["screen_name"],
11 >>>
"texto"
: dic["text"],
12 >>>
"fecha"
: fecha})
```

Para comenzar, generamos un objeto *Api()* que hará de interfaz con los web-services de Twitter. Este objeto dispone de muchos métodos muy útiles, pero no profundizaremos mucho en ellos



Figura 2: Nuestro programación en acción.

(es posible estudiarlos en [4]), sino que emplearemos directamente el método *getUserTimeline()*. Por defecto recupera 20 mensajes, pero podemos indicarle en un segundo parámetro cuántos queremos.

La lista que devuelve se compone de objetos *Status* que cuentan (de nuevo) con gran cantidad de métodos. Para simplificar nuestro trabajo empleamos el método *asADict()*, que nos devuelve

en segundos Unix (desde el 1 de enero de 1970), que podemos convertir en la fecha actual mediante el método *fromtimestamp()* de *datetime*. Por último almacenamos los datos que más nos interesan como un diccionario en una lista.

Ya tenemos tanto nuestros correos IMAP como los tweets que queremos controlar. Esto ha sido la parte fácil, ahora nos toca crear un sistema que

un diccionario con los datos que definen ese tweet. Uno de esos datos es la fecha, pero en formato de cadena de caracteres, lo que nos obligaría a realizar conversiones, por lo que hago uso del método *GetCreatedAtInSeconds()* para obtener la fecha

nos permita recoger esta información cada cierto tiempo y mostrarla.

Kronos

Aparte de ser el nombre del dios del tiempo, *kronos* [5] es una librería que Irmen de Jong (creador de la librería *Pyro*) ha extraído del proyecto *Turbo Gears* [6]. Irmen encontró de gran utilidad el sistema que *TurboGears* empleaba para la ejecución de tareas rutinarias en background. Y precisamente es eso lo que nosotros necesitamos; queremos poder ejecutar código que compruebe nuestro correo electrónico y ciertas cuentas de Twitter de forma periódica dentro de nuestro programa.

Debemos descargar *Kronos* y ponerlo en el mismo directorio en el que estamos trabajando. *Kronos* nos ofrece varias opciones para ejecutar las tareas. Podemos emplear *ThreadedScheduler*, que las ejecutaría en hebras dentro de nuestro programa, o *ForkedScheduler*, que haría lo mismo pero empleando procesos independientes. Por simplicidad utilizaremos la primera opción,

Listado 1: Mashup.py

```

001 import cherrypy
002 from kronos import *
003 from imaplib import IMAP4_SSL
004 import time
005 import re
006 import email
007 from email.header import
008 decode_header
009 from mako.template import
010 Template
011 from mako.lookup import
012 TemplateLookup
013 import twitter
014 from datetime import datetime
015 plantillas =
016 TemplateLookup(directories=['
017 plantillas'],
018 input_encoding='utf-8',
019 output_encoding='utf-8')
020 def fecha():
021 return
022 time.strftime(u"%Y-%m-%d a
023 las %H:%M:%S")
024 ## Datos globales
025 # No es la mejor idea, pero
026 funciona cuando una sola
027 hebra
028 # escribe y muchas leen.
029 Deberían estar bloqueados,
030 pero
031 # complicaría el código de
032 este artículo
033 datos_correos =
034 {"actualizado": fecha(),
035 "correos": [],}
036 datos_twitter = []
037
038 # Esta clase representa la
039 única página que tiene
040 # nuestra aplicación.
041 class MisCorreos:
042 def index(self):
043 plantilla =
044 plantillas.get_template("inde
045 x.html")
046 return
047 plantilla.render_unicode(corr
048 eos = datos_correos, twitter
049 = datos_twitter)
050 index.exposed = True
051
052 class TareaTwitter:
053 def __init__(self):
054 self.api = twitter.Api()
055
056 def __call__(self,
057 usuarios):
058 global datos_twitter
059 datos_twitter =
060 self.get_twitter(usuarios)
061
062 def get_tweets(self,
063 usuario):
064 """Obtiene los tweets de un
065 usuario"""
066 lista =
067 self.api.GetUserTimeline(usua
068 rio)
069 tweets = []
070
071 for tweet in lista:
072 dic = tweet.AsDict()
073 segundos =
074 tweet.GetCreatedAtInSeconds()
075 fecha =
076 datetime.fromtimestamp(segund
077 os)
078 tweets.append({"nombre":
079 dic["user"]["screen_name"],
080 "texto": dic["text"],
081 "fecha": fecha})
082 return tweets
083
084 def get_twitter(self,
085 usuarios):
086 """Se conecta por IMAP al
087 servidor de correo
088 y comprueba si hay
089 correos"""
090 tweets = []
091 for usuario in usuarios:
092 tweets +=
093 self.get_tweets(usuario)
094
095 tweets = sorted(tweets,
096 key=lambda x: x['fecha'],
097 reverse=True)
098 return tweets
099
100 class TareaCorreo:
101 def __init__(self,
102 servidor, usuario, clave):
103 self.servidor = servidor
104 self.usuario = usuario
105 self.clave = clave

```

```
>>> from kronos import 2
ThreadedScheduler
>>> tareas =
ThreadedScheduler()
```

A este *scheduler* podemos pasarle el código a ejecutar mediante una función. Como la recolección de los datos puede ser compleja, es mejor crear clases *callable*, clases que incorporan el método `__call__`. Podemos crear objetos a partir de ellas que se comportarán como funciones, y por tanto, podremos pasar al *scheduler*. El usar clases nos permitirá estructurar mejor nuestro código, como se muestra en las clases *TareaTwitter* y *TareaCorreo* del Listado 1.

Siguiendo con el Listado 1, si vamos al final veremos cómo podemos añadir las tareas al *scheduler* empleando el método `add_interval_task()`. El primer parámetro es la función a llamar, el segundo el nombre de la tarea (para errores y log), el cuarto el retraso inicial antes de ejecutarla, el quinto es el intervalo en segundos en que queremos que se ejecute la función. Los tres últi-

mos parámetros son el método (en esta caso mediante una hebra), los argumentos para la función y los argumentos con nombre de la función. Por si esto ha despistado al lector, una función en Python puede tener argumentos normales, como por ejemplo `funcion(1,"Hola",[])`, o argumentos con nombre, como en `funcion(numero=1, texto="Hola", nombres=[])`.

Kronos también posee un método llamado `add_daytime_task()` que nos permite ejecutar una tarea a una hora en particular del día.

Una vez cargadas las tareas, podemos arrancar el *scheduler* con el método `start()`, comenzando a ejecutarse las tareas inmediatamente, ya que en el Listado 1 hemos indicado que el retraso inicial sea de 0 segundos. Estas tareas se ejecutan como hebras en segundo plano, por lo que nuestro programa seguirá procesándose.

Hay que tener cuidado en poner intervalos de ejecución más largos que el tiempo que necesita nuestro programa para recopilar la información. Como cada sistema y conexión a Internet es

distinta, lo mejor es hacer pruebas antes de utilizar un intervalo demasiado corto. Si lo fuera, puede que el programa comience a volver a recoger datos antes de que haya acabado la primera recopilación. Es mejor pasarse en el intervalo de ejecución que emplear uno probablemente demasiado corto.

Nuestras tareas y el *scheduler* que las ejecutarán ya están listos, sólo necesitamos la aplicación sobre la que trabajar.

CherryPy

Siento cierta predilección por CherryPy [7], un proyecto que considero ha sido muy subestimado. Nunca me cansaré de cantar las bondades de un servidor web, con un rendimiento más que aceptable, que con sólo 5 líneas de código nos permite poner online código Python. Después de toda esta propaganda gratuita de CherryPy, es hora de demostrar su poder.

Nuestra aplicación es capaz de recopilar los datos que necesitamos, pero ahora vamos a tener que inventarnos algún método de comunicárselos a CherryPy. Emplearemos uno muy tosco

Listado 1: Mashup.py (Cont.)

```
071 def __call__(self):
072     global datos_correos
073     datos_correos =
074         self.comprueba_correo()
075 def get_header(self,cadena):
076     """Procesa una cabecera de
077     un email """
078     cadena =
079         decode_header(cadena)[0][0].d
080         ecode('iso8859-1')
081     return unicode(cadena)
082 def conecta_imap(self):
083     """Devuelve conexión con
084     server IMAP"""
085     con =
086         IMAP4_SSL(self.servidor)
087     print
088     con.login(self.usuario,
089             self.clave)
090     return con
091 def procesa_correo(self,
092     mensaje):
093     """Procesa el correo
094     devolviendo un diccionario
095     con
096     los datos importantes."""
097     msg =
098         email.FeedParser.FeedParser()
099     msg.feed(mensaje[0][1])
100     datos = msg.close()
101     correo = {"fecha":
102             self.get_header(datos['Date']
103             ), "desde":
104             self.get_header(datos['From']
105             ), "asunto":
106             self.get_header(datos['Subjec
107             t'])}
108     return correo
109 def comprueba_correo(self):
110     """Se conecta por IMAP al
111     servidor de correo
112     y comprueba si hay
113     correos"""
114     datos = {}
115     datos["actualizado"] =
116         fecha()
117     datos["correos"] = []
118     con = self.conecta_imap()
119     con.select('INBOX')
120     # Consultamos solo aquellos
121     correos que vengan de
122     linux-magazine
123     status,lista_mensajes =
124     con.search(None,'FROM',"linux
125     -magazine")
126     try:
127         for id_mensaje in
128             lista_mensajes[0].split():
129             cmd =
130             "(BODY[HEADER.FIELDS (SUBJECT
131             DATE FROM))"
132             respuesta, mensaje =
133             con.fetch(id_mensaje, cmd)
134             correo =
135                 self.procesa_correo(mensaje)
136             datos["correos"].append(correo)
137         finally:
138             con.close()
139             print con.logout()
140     # Damos la vuelta a los
141     correos
142     datos["correos"] =
143     list(reversed(datos["correos"]
144     ))
145     return datos
146 if __name__ == "__main__":
147     usuarios =
148     [u"RCarpintier",u"linux_spain
149     "]
150     tareaCorreoImap =
151     TareaCorreo( servidor =
152     "el.servidor.com", usuario =
153     "nombre-usuario", clave =
154     "clave-correo")
155     s=ThreadedScheduler()
156     s.add_interval_task(
157     tareaCorreoImap, "Correos",
158     0, 120, method.threaded,
159     None, None)
160     s.add_interval_task(
161     TareaTwitter(), "Twitter",
162     0, 120, method.threaded,
163     [usuarios], None)
164     s.start()
165     cherrypy.quickstart
166     (MisCorreos())
```

y sencillo, pero que debido al diseño que estamos usando, va a funcionar bastante bien. La idea es emplear dos variables globales, una para Twitter y otra para IMAP, que almacenarán los datos recogidos por TareaTwitter y TareaCorreo. Debido a que sólo habrá una hebra escribiendo en cada una de estas variables, y que CherryPy se limitará a leer su contenido, no experimentaremos problemas al trabajar con ellas. Si leemos el código del Listado 1, las variables `datos_correos` y `datos_twitter` serán las encargadas de almacenar los datos recogidos. La configuración de CherryPy es tan pequeña, que incluso podemos echarle un vistazo (ver *MisCorreos* en Listado 1).

CherryPy sólo necesita una clase, y enlaza sus métodos con urls. En este caso, el método `index()` se enlaza con la url `/index.html`. La configuración por defecto de CherryPy emplea el puerto 8080, sólo tenemos que cargar `http://localhost:8080`, y veremos nuestro programa en funcionamiento.

Mako

El último elemento de nuestro sistema es la infraestructura de plantillas *Mako*

[8]. En el Listado 2 aparece la sencilla plantilla que hemos empleado para generar la página web que se muestra en la Figura 2.

Si observamos la variable `plantillas` en el Listado 1, veremos que configuramos *Mako* para que busque un directorio llamado `plantillas` en el mismo directorio en el que esté nuestro programa. En el anterior ejemplo de código de CherryPy podemos ver cómo en el método `index()`, lo único que hacemos es cargar la plantilla `index.html` y generar el html resultante mediante el método `render_unicode()` y los datos que han recopilado las dos tareas.

¡Listo!, tenemos un maravilloso «mash-up» como cualquiera de esas empresas modernas Web 2.0. Es verdad que el nuestro es más simple, pero también es cierto que podemos mejorar la plantilla como queramos, podemos añadir nuevas tareas, y que esta funcionalidad la hemos conseguido con un pequeño programa Python, que, sin embargo, incorpora un servidor web, se comunica con IMAP, se comunica con Twitter y emplea un sistema de gestión de tareas. ¡Todo en uno!

Conclusión

Gran parte de la potencia de Python está en sus librerías. Hay que aprender a buscarlas y a utilizarlas para no reinventar la rueda cada vez que creemos un programa. Es muy recomendable pasarse por Pypi [9] antes de ponernos a escribir código fuente, es probable que alguien ya haya resuelto nuestro problema. Es importante tomar el control ante la avalancha de información que recibimos diariamente. Espero que el lector encuentre útil este pequeño programa y que lo mejore y adapte a sus necesidades... ¡antes de acabar sepultado bajo montañas de bits!

RECURSOS

- [1] PyCon: <http://us.pycon.org>
- [2] imaplib: <http://docs.python.org/library/imaplib.html>
- [3] Twitter: <http://www.twitter.com>
- [4] Python-twitter: <http://code.google.com/p/python-twitter/>
- [5] Kronos: <http://www.razorvine.net/download/kronos.py>
- [6] TurboGears: <http://turbogears.org/>
- [7] CherryPy: <http://www.cherrypy.org>
- [8] Mako: <http://www.makotemplates.org/>
- [9] Pypi: <http://pypi.python.org/pypi>

Listado 2: index.html

```

01 ## -*- mode: html; coding:      26      }
    utf-8 -*-                      27
02 <html                          28     table tr {
    xmlns="http://www.w3.org/1999/  29     vertical-align:top;
    xhtml" xml:lang="es"           30     }
    lang="es">                      31     table td {
03 <head>                          32     border: 1px solid #ccc;
04 <title>Simple Option: Tu        33     }
    tienda online</title>          34
05 <meta                            35 </style>
    http-equiv="content-type"      36
    content="text/html;"           37 </head>
    charset=utf-8" />             38 <body>
06 <style>                          39
07     body {                       40 <div id="contenido">
08     background: #eee;           41 <h1>Correos </h1>
09     margin: 0;                  42
10     padding: 0;                 43 <p>Actualización:
11     }                            44     <strong>${correos["actualizado
12                                 45     "]}</strong></p>
13     div {                       46     <table>
14                                 47     <tr class="principal">
15     font-size: 10px;            48     <td>
16     font-family: verdana;       49     <tr>
17     width: 80%;                  50     <th>Nombre</th>
18     margin: 0;                   51     <th>Texto</th>
19     padding: 0;                  52     <th>Fecha</th>
20     margin: 0 auto;              53     </tr>
21     background: white;          54 %for tweet in twitter:
22     }                            55 <tr>
23                                 56 <td>${tweet["nombre"]}</td>
24     table {                       57 <td>${tweet["texto"]}</td>
25     font-size: 100%;            58 <td>${tweet["fecha"]}</td>
26                                 59 %endfor
27                                 60 </table>
28                                 61 </td>
29                                 62 <td>
30                                 63
31                                 64 <table id="correos">
32                                 65 <tr>
33                                 66 <th>Asunto</th>
34                                 67 <th>Fecha</th>
35                                 68 <th>De</th>
36                                 69 </tr>
37                                 70
38                                 71 %for correo in
39                                 72     correos["correos"]:
40                                 73     <tr><td>${correo["asunto"]}</t
41                                 74     d><td>(${correo["desde"]})</td
42                                 75     > <td>
43                                 76     ${correo["fecha"]}</td></tr>
44                                 77     %endfor
45                                 78 </table>
46                                 79
47                                 80 </div>
48                                 81
49                                 82 </body>
50                                 83 </html>

```