Python

ENTRE TODOS

Old No.5 BRAND

QUALITY

Community

PROGRAMMING

MAGAZINE

march2012 40% Vol.

Revista de la comunidad Python Argentina

http://revista.python.org.ar/

ISSN: 1853-2071

Licencia

En Este Número

Staff

Editores: Juan Bautista Cabral - Tomas Zulberti

Sitio: http://revista.python.org.ar

Editor responsable: Roberto Alsina, Don Bosco 146 Dto 2, San Isidro, Argentina.

ISSN: 1853-2071

PET es la revista de PyAr, el grupo de usuarios de Python Argentina.

Para aprender sobre PyAr, visite su sitio: http://python.org.ar Los artículos son (c) de sus respectivos autores, reproducidos con autorización. El logo “solpiente” es creación de Pablo Ziliani. La imagen de la portada fue hecha por Juan B Cabral.

BUSCADO: Organizador de la próxima PyConar

Autor: Juan Rodriguez Monti
Web: http://www.juanrodriguezmonti.com.ar/
Email: juanrodriguezmonti@gmail.com
Twitter: @jrodriguezmonti

PyConAr.2011

Durante los primeros meses del año se definió que PyConAr 2011 se haría en Junín (Buenos Aires) y que yo seria el coordinador general. En ese momento me sentí tentado de pensar que faltaban unos 10 meses para el evento y que era mucho. Pero no lo hice, y hoy les puedo decir que se requiere de tiempo para hacer todo bien y llevar adelante un evento tan importante.

Y si usted, lector quiere organizar un evento de estas características; ¿qué es lo que tiene que hacer?. Bueno... masomenos esto:

El primer paso a dar, es conseguir un buen lugar. Esto es: que entren entre 300 y 500 personas, que haya una conexión Wi-Fi aceptable, que sea cómodo, que haya lugares cerca en donde comer, y que el acceso sea fácil para nadie sienta como un lugar tan ignoto pudo llegar hospedas a PyConAr.

Luego, tenés que elegir día. Buscá en Internet las fechas ya pautadas de eventos a los que vayan a asistir geeks y tenelos en cuenta. Tratá, por favor, de ponerte en contacto con organizadores de eventos similares (programación, seguridad informática, internet, etc) que tengan público en común, para evitar así que se superponga nos beneficiemos todos En nuestro caso hablamos con la gente de Python Brasil, y establezcimos fechas cercanas para que sea menos costoso movilizar a los invitados internacionales.

Es útil que durante los primeros meses de la planificación armes la estructura del programa del evento:

- Estimá la cantidad total de charlas.
- Definí si va a haber 2, 3, 4 charlas simultáneas.
- Armate un boceto de los dos días o tres de evento.

No importa qué charla ubicas en qué lugar, eso después se puede cambiar.

Una vez que tenés armado el cronograma, sentate a tomar un café con la gente del lugar elegido para PyConAr, y reservá los salones que necesites, los cañones, y todos los recursos que quieras.

De ser posible 5 o 6 meses antes del evento tendrías que tener lista de distribución de correo aparte.

Cuando todo lo anterior está en marcha, empezá a pensar en los invitados internacionales. El monto de inversión de Sponsors que hayas conseguido, junto con el dinero que vas a poder compartir con Python Brasil u otra comunidad, van a ser las dos principales fuentes de ingreso que permitirán definir la cantidad de invitados internacionales que podrás tener.

No te olvides que si bien hay mucha gente brillante afuera, también la hay acá. No te dejes encandilar por los flashes de un nombre ilustre cuando también hay tipos increibles en Argentina. Es decir: invitá gente de afuera, pero sabiendo que también acá hay gente igual o más capaz. Aprovechá lo que unos pocos kilómetros a la redonda tuya te da y no te deprimas si no podés traer más de un invitado internacional. La riqueza del evento tal vez pase por otro lado.

Servite de las redes sociales para hacer ruido con el evento. Twitter ha sido en nuestro caso (@PyconAr (http://twitter.com/PyconAr/)), un recurso valiosísimo para difundir noticias; por otro lado si necesitás hacer una nota más larga, el blog de PyConAr tenes disponible (http://pyconar.blogspot.com/) o ,para otros casos; se puede usar la wiki de Python Argentina (http://python.org.ar/pyar/) si, por ejemplo, deseas tener en la web el contenido de un email como el call for charlas.

¡Hacé merchandising!. A los geeks nos encanta, y la colaboración de Sponsors para PyConAr permite hacer algunos regalos a la gente!.

Invitá a cuanta empresa se te atraviese a participar como Sponsor, y no pienses: “A estos tipos de Estados Unidos o Inglaterra o Finladia no los invito porque me van a ignorar”; por que gracias a eso conseguimos apoyo de empresas de afuera. Asi que, navegar por los sitios de otras PyCon y ver los sponsors, puede ser util para invitarlos a participar de PyConAr.

¿Vivís en una ciudad en donde no hay empresas que quieran participar como sponsor?: ¡Qué importa!; PyConAr es lo suficientemente grande como para atraer sponsors de todo el mundo, así que escribí mails a todos lados ni bien se confirme que vas a ser el organizador.

Otro dato: establecé fechas y tratá de respetarlas. Podés atrasarte un poco, o cambiar algo la oferta, pero es importante dar una imagen seria del evento para que el público se interese y te respete; con esto también logras ser confiable para los sponsors y los

disertantes.

Tratá de filmar todas las charlas aunque no te alcance para lograr videos de calidad profesional, con cuatro trípodes comprados en una página internacional que te manda gratis los productos, y 3 ó 4 cámaras estarás listo para tener un registro invaluable del evento sobre todo para gente no pudo asistir.

Hacé un regalo a los invitados (disculpá el tono imperativo, pero son solo sugerencias, sentite libre de tirar todo lo que sugiero a /dev/null) que esté relacionado con el lugar y con el evento (un detalle como “PyconAr” grabado es un detalle excelente).

Es importante que estés en contacto permanente con Python Argentina ya sea por el IRC o la lista de correo; y entre chat y mail podes mojar la garganta con unos mates, una legui, un vodka o un Jack Daniel’s.

Por último, tratá de planificar las cosas para que un mes y medio antes de la fecha del evento esté todo listo. Todo quiere decir:

- Todo lo que necesites de los proveedores ya recibido.
- Los proveedores del evento pagos.
- Los disertantes y las charlas confirmadas.
- El lugar listo
- Todo TODO **TODO** listo

¿Por qué esa desesperación?: Te puedo asegurar que van a surgir cosas que no planeaste, y ese mes y algo es infinitamente útil. Yo hice esto y pude resolver algunos asuntos de último momento en esos días de backup que me reservé.

Como me dijo el organizador de PyConAr 2010, Horacio "perrito" Durán: **"Delegá todo lo que puedas"**; yo lo cito y adhiero a que lo hagas. **¡Delegá!**.

Supongo que es obvio, pero lo digo igual: Armá un grupo de trabajo en tu ciudad. No es necesario que todos sean programadores ni sean informáticos mientras sean gente de tu confianza para que te ayude.

Te sugiero, también, que los primeros días de la planificación definas un presupuesto estimado y que en base a ese armar dos o tres opciones organizativas basadas en la cantidad de dinero recaudado.

Es deseable también que te sirvas del sitio web y centralices recursos informativos allí.

Pedí ayuda cuando necesites, ¡Todo PyAr estará para ayudarte!.

Así que... ¿No te postularías para ser organizador general de una PyConAr enviándome un mail?.

Gracias a todos (equipo de trabajo, PyAr, público, disertantes, sponsors, etc) los que hicieron posible PyConAr 2011. Gracias a todos por la buena onda, también. A modo de cierre: Lo más importante de un evento, en mi humilde opinión, es que cada asistente salga inspirado y con ganas de aprender más y de mejorar y nutrir su inteligencia tanto como pueda

¡Larga vida a PyConAr, a PyAr y a la pitón!.

Desarrollo de aplicaciones móviles para Android con Python

Autor: Tatiana Al-Chueyr

Bio: Ingeniera de computación, programa con Python desde 2003. Fue creadora de los softwares libres InVesalius 3 y Molecular-View, hechos con wxPython y VTK. Hoy día desarolla con Python y Django en Globo.com, estudiando desarollo para Android en su tiempo libre. También fue creadora de la conferencia brasileña AndroidConf.

Web: http://www.tatialchueyr.com/

Email: tatiana.alchueyr@gmail.com

Twitter: @tati_alchueyr

Introducción a SL4A

El desarrollo de aplicaciones móviles para Android se hace generalmente utilizando Java como lenguaje de programación. Lo que muchos pythonistas no saben es que desde Junio de 2009 es posible escribir aplicaciones para Android en Python empleando SL4A. (Anuncio de Google [1]).

SL4A (Scripting Layer for Android) [2], originalmente llamado de ASE (Android Script Enviroment), permite que desarrolladores editen, ejecuten scripts y interactúen con intérpretes directamente en el dispositivo Android. Estos scripts tienen acceso a muchas de las APIs del sistema operativo (SO), pero con una interfaz muy simplificada que hace que sea fácil:

- Realizar llamadas telefónicas
- Enviar mensajes de texto
- Escanear códigos de barras
- Obtener ubicación y datos de los sensores
- Usar texto-a-voz (text-to-speech / TTS)

y mucho más

El proyecto fue empezado por Damon Kohler [3] y, así como la mayor parte de Android, SL4A es open-source y es distribuido bajo la licencia Apache 2.0. SL4A también soporta otros lenguajes, como Beanshell, JRuby, Lua, Perl y Rhino. Pero, en ese artículo veremos solamente el uso de SL4A para Python.

Cómo instalar SL4A

Para instalar SL4A, hay que habilitar la opción “Fuentes desconocidas” (“Unknown sources”) en las configuraciones de “Aplicación” (“Application”) de su dispositivo.

Al momento de escribir este artículo, SL4A tiene la calidad de un software alpha, por eso no está disponible en Android Market. Para instalarlo, puede hacer el download del apk desde el sitio oficial [4] o por el siguiente barcode:

Después de hacerlo, debe instalar el interprete Python para SL4A, a través del link [5] o del barcode de abajo:

Intérprete Python en SL4A

La versión más reciente de Python para Android es una compilación del CPython 2.6.2.

Para interactuar con el interprete, debe:

1. Elegir SL4A de la lista de aplicaciones
2. En SL4A, presionar el botón “menú” y elegir la opción “View”
3. Eligir “Interpreters” y, de la lista, seleccionar “Python”. Uno verá algo como:

Entonces, se puede escribir, por ejemplo:

```
>>> import sys
>>> sys.platform
```

Y el interprete python mostrará algo como:

```
'linux-arm-71'
```

Con ese código vemos que python de SL4A fue compilado para la arquitectura ARM, procesador utilizado en la mayoría de los smartphones de hoy.

Módulos disponibles

Como uno puede ver, el intérprete es como el de las PCs, con diversos módulos de la librería estándar (standard library) como: *glob*, *httplib*, *math*, *os*, *pickle*, *shlex*, *shutil*, *ssl*, *string*, *subprocess*, *sys*, *tempfile*, *time*, *thread*, *unittest*, *urllib*, *uuid*, *xml*. Hasta el Zen del Python, de Tim Peters, es disponible (*import this*).

También está el módulo *gdata*, con lo cual se puede tener acceso a la interfaz de servicios de Google, como crear y editar spreadsheets.

Si usted ya desarrolló módulos con Python puro y le gustaría usarlos en el Android, no hay problema, es posible adicionarlos en el directorio del teléfono

/sdcard/com.googlecode.pythonforandroid/extras/python. Con lo que será posible usarlos como si fueron parte de la biblioteca estándar.

Módulos en archivos *egg* también son soportados a partir del release 4 de python-for-android, pero es necesario seguir algunos pasos para hacerlos [6].

Ya hay paquetes hechos para algunas librerías útiles [7], como: - **PyBluez**, para interactuar con bluetooth - **Twisted**, engine para red, basada en eventos - **Zope**, interfaz para Zope - **pyEphem**, para datos astrológicos

La comunidad también ya ha portado la conocida librería PyGame para Android: - **PyGame** [8]

Así, si uno quiere jugar con proyectos hechos con PyGame para su computadora en el Android, ya es posible hacerlo con SL4A!

Primer ejemplo - Hello World

El intérprete Python de SL4A ya viene con algunos ejemplos de como usar la API de Android con Python.

Para ejecutar el primer ejemplo, debes: 1. Abrir la aplicación SL4A #. Elegir el script *hello_world.py* #. Escoger el lápiz (editar) #. Mirar el código abajo:

hello_world.py

```
import android
droid = android.Android()
droid.makeToast('Hello, Android!')
print 'Hello world!'
```

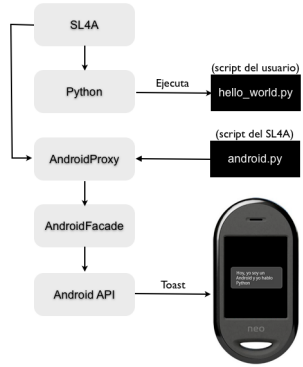
5. Cambiar *Hello, Android!* pasado como parámetro para el método *makeToast* por *Hoy, yo soy un Android y yo hablo Python*
6. Desde el botón *Menu*, elegir *Save & Run*
7. Ver la exhibición del diálogo con *Hoy, yo soy un Android y yo hablo Python*

Que se pasa adentro del android con SL4A

En su nivel más bajo, SL4A es un host de scripting. De un lado, hay un intérprete Python en C compilado para la arquitectura ARM, usando Android Native Development Kit (NDK). De otro, está el sistema operativo Android, al cual uno puede tener acceso por el mecanismo RPC (Remote Procedure Call) y JSON (JavaScript Object Notation).

SL4A implementa el archivo *android.py*, lo cuál define una serie de funciones proxy necesarias para comunicarse con la API de Android OS. Con eso, el haga la interfaz entre el intérprete (python) y Android OS (servidor), pasando comandos e informaciones. Fue hecho así para prevenir que cualquier script malicioso hiciera algo perjudicial al OS.

El diagrama de Figura 5 ejemplifica la ejecución de SL4A, y es basada en el libro [9].



Segundo ejemplo - envío de SMS con coordenada GPS

Imagine que usted está manejando para ir a cenar con su novia, pero hay un gran atasco de tráfico. Como hay policías en la cercanía, no puedes llamarla mientras maneja.

Uno puede desarrollar un script en Python muy sencillo que verifique su ubicación (con el GPS de su Android) y lo envíe al teléfono móvil de su novia por mensaje (SMS).

envio_gps_por_sms.py

```
import android
from time import sleep

droid = android.Android()
droid.startLocating()
sleep(15)
latitud = droid.readLocation().result["network"]["latitude"]
longitud = droid.readLocation().result["network"]["longitude"]
droid.stopLocating()

telefono = "541136892092"
mensaje = "Cariño, mucho trafico. Estoy en http://maps.google.com/maps?q=%s,%s" % (latitud, longitud)
droid.smsSend(telefono,mensaje)
```

Algunos posibles cambios en ese script:

- cambiar el número para donde va enviar el SMS por diálogo (*541136892092*), por:

```
telefono = droid.dialogGetInput(" GPS to SMS", "Phone number:")
```

- O, si no quisiera enviar la localización para alguien, pero quisiera ver donde está en un mapa, uno podría remover el código después de *droid.stopLocating()* y escribir solamente:

```
droid.webViewShow('http://maps.google.com/maps?q=%s,%s' % (latitud, longitud))
```

Escribiendo el código de Android en una PC

Como uno puede ver, no es muy práctico escribir código utilizando el teclado de Android, mismo empleando recursos que facilitan como Swype [10].

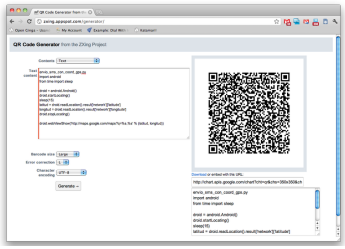
¿No sería más cómodo escribir el código en la PC y enviarlo a su smartphone?

Hay varias maneras de hacerlo. Algunas de ellas:

A. Códigos de barras (barcodes)

Uno puede escribir el código en la computadora y, a partir de esta, generar un barcode. Entonces, se puede importarlo leyéndolo con la cámara del Android.

De su computadora: 1. Escriba el código fuente #. Acceda: http://zxing.appspot.com/generator/ #. En el sitio, elija “Text” (en “Contents”) #. En “Text content”, la primer línea debe de ser el nombre de su archivo python (ejemplo: *envio_sms_con_coord_gps.py*). Las demás deben contener el código del último ejemplo. #. Presione el botón “Generate”



Después de obtener el barcode, uno debe abrir SL4A, apretar el botón “menú”, elegir “Add” y, en el menu, elegir “Scan barcode”. Entonces, debe poner la cámara del

android mirando a la imagen que generó. Con eso, uno puedrá ver su programa en la lista de programas de SL4A.

B. Envío de archivos con ADB

Para eso, debe bajar/instalar Android SDK (Software Developer Kit), como se detalla en el sitio oficial [11].

Después, puede tener acceso al ADB (Android Debug Bridge) en el directorio: *android-sdk/platform-tools*

Es mejor es poner *android-sdk/platform-tools/adb* en el *$PATH* (por ejemplo en el *.bash_profile*), para facilitar su uso.

Conecte su teléfono a la computadora con un cable USB.

Hecho eso, podrá enviar archivos para su teléfono, por el terminal, con:

```
$ adb wait-for-device
$ adb push myscript.py /sdcard/sl4a/scripts
```

Consulte la documentación de *adb* para obtener más informaciones sobre estos y otros comandos.

C. Control del smartphone con ADB

Podría controlar su Android como si fuera un servidor, a partir de su computadora.

Para eso, debe:

1. De Android: De *Menu* en SL4A, elegir *View* y *Interpreters*. Entonces, en *Menu*, elegir *start server* y *private*. En la barra de notificaciones (arriba de la pantalla), uno deber elegir *SL4A Service* y ver el puerto donde el servidor está (ejemplo: *4321*). Conectar el android a la computadora, con el cable USB
2. De la computadora, ejecutar:

```
$ adb start-server
$ adb forward tcp:9999 tcp:4321
$ export AP_PORT=9999
```

Entonces, usted debe bajar el archivo *android.py* [12] y ponerlo en su *$PYTHONPATH*.

Entonces, uno podrá ejecutar los ejemplos de arriba, a partir de su PC, viendo las respuestas en su android.

D. Otras maneras

Podrá también usar un emulador android en su computadora, para después enviar el código a su teléfono. También se puede enviar el código por email, o ssh, o cualquier otro servicio.

App python en android con interfaz gráfica

Es posible emplear la interfaz web para desarrollar aplicativos para Android con Python, a través de webview.

El próximo ejemplo muestra como el script python llama webview, que recibirá palábras y las enviará al script python, para hacer hablar el android.

Observación: hay que aumentar el volúmen para escuchar las palabras.

webview_text_to_speech.py

```
import android
droid = android.Android()
droid.webViewShow('file:///sdcard/sl4a/scripts/html/text_to_speech.html')
while True:
  result = droid.waitForEvent('say').result
  droid.ttsSpeak(result['data'])
```

text_to_speech.html

```
<html>
  <head>
    <title>Text to Speech</title>
    <script>
      var droid = new Android();
      var speak = function() {
     droid.eventPost("say", document.getElementById("say").value); }
    </script>
  </head>
  <body>
    <form onsubmit="speak(); return false;">
      <label for="say">que te gustaria hablar?</label>
      <input type="text" id="say" />
      <input type="submit" value="Speak" />
```

```
    </form>
  </body>
</html>
--
```

Se puede mejorar la interfaz con CSS. Con webview, la interfaz de la aplicación Python pasa a ser un aplicativo web como cualquier otro.

Una alternativa a escribir aplicaciones GUI para Android usando Python es combinar SL4A a PySide para Android. Con eso permitirá escribir interfaces de usuario Qt o QML para Android todo en Python puro [13]. Pero esto solo sería tema para otro artículo.

Otros métodos de la API de Android en SL4A

Después de llamar:

```
import android
droid = android.Android()
```

Hay muchos comandos nativos de Android que se puede emplear. Vimos *makeToast*, *ttsSpeak*, *readLoaction* y *smsSend*, por ejemplo. Pero hay muchos otros [14]. Algunos otros ejemplos:

```
droid.vibrate(100)
droid.batteryGetLevel()
droid.cameraCapturePicture()
droid.scanBarcode()
droid.search('alguna cosa')
droid.viewContacts()
droid.sensorsReadAccelerometer().result
```

Además, es posible también enviar correos, gravar audio, tener acceso la interfaz del teléfono, enviar SMS, reconocer speeches, tener acceso al wifi, sensores o usar diálogos padrones de Android.

Como compartir aplicativos hechos con SL4A

Una vez que hecha la aplicación con SL4A y Python, uno puede compartirla con amigos. La manera más sencilla es compartir el barcode. Pero si sus amigos no son desarrolladores, también se pueden crear paquetes ejecutables (apk) 'standalones', como detallado en el sitio oficial [15].

Utilización

Algunas referencias [16] informan que SL4A debe ser usado solamente:

- RAD (rapid application development, o desarrollo rápido de aplicaciones) para crear prototipos para ensayar la viabilidad de una idea, de modo que después uno pueda crear una aplicación en toda regla en java para Andorid
- Escritura de testeo, en el supuesto de que el apoyo de la API de Android está expuesto a SL4A, que puede ser utilizado para probar los scripts para otras funcionalidades
- Construcción de servicios públicos, para hacer trabajos pequeños o automatizar las tareas repetitivas que no requieren interfaces complejas

Pero yo creo que la limitación del uso debe de ser solamente la creatividad de los desarrolladores.

¡Buen hacking con Python y SL4A en Android!

Más ejemplos de uso de SL4A con Python

- Android Twisted SpyCam: http://kbcarte.wordpress.com/2011/08/31/android-twisted-spycam/
- Send random Chuck Norris jokes: http://www.h3manth.com/content/sms-android-using-python
- Android sensors: https://github.com/Jonty/RemoteSensors/
- Twitter client: http://www.linux-mag.com/id/7370/
- Cellbot controller: https://github.com/georgegoh/cellbot-controller
- Envío de SMS a partir del linux: http://ernestocrespo.blogspot.com/2011/06/program-for-sending-sms-from-linux-to.html
- Android, Python and Arduino para automatización del hogar: https://github.com/tatiana/droiduino

Referencias

1 http://google-opensource.blogspot.com/2009/06/introducing-android-scripting.html *“Anuncio del lanzamiento de ASE, hoy día conocido por SL4A”*

2 http://code.google.com/p/android-scripting *“Sítio oficial de SL4A, en Google Code”*

3 http://www.damonkohler.com/ *“Blog de Damon Kohler, creador de SL4A”*

4 http://android-scripting.googlecode.com/files/sl4a_r4.apk *“Última versión SL4A, 15 de Noviembre de 2011”*

5 http://android-scripting.googlecode.com/files/PythonForAndroid_r4.apk *“Última versión PythonForAndroid, 15 de Noviembre de 2011”*

6 http://code.google.com/p/python-for-android/wiki/BuildingModules *“Construcción de módulos en Python para Android”*

7 http://code.google.com/p/python-for-android/ *“Python for Android, en Google Code”*

8 http://pygame.renpy.org/ *“PyGame para Android”*

9 Pro Android Python with SL4A, Paul Ferrin, 2011, Apress

10 http://swype.com *“Swype, mejoras en el uso del teclado de Android”*

11 http://developer.android.com/sdk *“SDK Android”*

12 http://android-scripting.googlecode.com/hg/python/ase/android.py *“Módulo android.py para llamar SL4A de la computadora”*

13 http://thp.io/2011/pyside-android/ *“Uso de QT para hacer la interfaz gráfica de apps Python de Android”*

14 http://code.google.com/p/android-scripting/wiki/ApiReference *“API de referencia para aceso a el Android SO a partir de Python”*

15 http://code.google.com/p/android-scripting/wiki/SharingScripts
“Creación de APK para apps hechas con Python”

16 Practical Android Projects, Jucas Jordan and Pieter Greyling, Apress

Python y Arduino

Autor: Álvaro Justen

Bio: también conocido como Turicas, es usuario y activista del software libre desde hace más de 8 años; desarrollador Python; entusiasta Arduino y apasionado de Agile.

Blog: http://blog.justen.eng.br

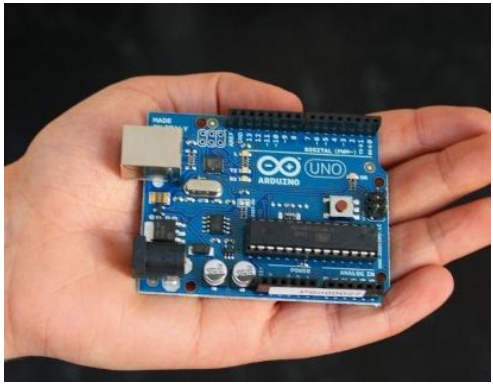
Twitter: @turicas

Email: alvaro@CursoDeArduino.com.br

Introducción

Arduino es una plataforma que facilita el desarrollo de proyectos de microcontroladores, que consta de dos partes: hardware y software. El hardware es una placa tan pequeña que cabe en la palma de la mano, está basada en la familia de microprocesadores AVR y tiene todas las especificaciones electrónicas abiertas. El software, por otra parte, es una IDE en conjunto con las librerías que se utilizan para usar la placa (ambas son libres).

Utilizando el Arduino es posible desarrollar proyectos de robótica (incluídos software, electrónica y mecánica) de una forma sencilla y rápida. Además, cómo el proyecto Arduino es abierto, hay muchos módulos de hardware y librerías disponibles, eso hace que el trabajo que tenemos que desarrollar para un proyecto es mayormente recopilar los módulos necesarios y hacerlos trabajar juntos.



La placa tiene un microprocesador de 16MHz, dispone de 32kB de memoria para almacenar el software de control y 2kB de RAM. Parece poco, pero es más que sufiente para el control de los pequeños dispositivos electrónicos, lectura de sensores y toma de decisiones.

Después de esta introducción nos ponemos manos a la obra.

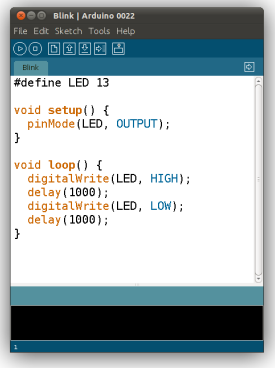
Hello world

Vamos a empezar por mostrar un ejemplo simple de código para Arduino: encendido y apagado de un LED (esto se conoce como el "hola mundo" de la informática embebida) - el lenguaje que usamos es un subconjunto de C++:

```
#define LED 13

void setup() {
    pinMode(LED, OUTPUT); //configures digital port 13 as output
    //in the Arduino board we have a LED connected to digital port 13
}

void loop() {
    digitalWrite(LED, HIGH); //turns LED ON
    delay(1000); //wait for 1000ms = 1s
    digitalWrite(LED, LOW); //turns LED OFF
    delay(1000); //wait for 1000ms = 1s
}
```

Después de introducir el código, simplemente haga clic en el botón "Upload" en el IDE, de esta manera son compilados y enviados a través de USB a la placa. Después

de este proceso, el software ya se está ejecutando en la placa, y ya no necesitaremos la pc para que funcione.



Las funciones utilizadas en este código (*pinMode*, *digitalWrite* y *delay*) ya están definidos en la librería estándar de Arduino (al igual que muchas otras) y por lo tanto no hay necesidad de importación, el IDE ya lo hace automáticamente (por esto el "lenguaje" se llama un subconjunto de C++).

El código anterior se puede traducir como:

- Configurar el puerto 13 como puerto de salida digital, es decir, que control de tensión (0V o 5V) se realiza mediante el software
- Coloque en el puerto 13 5V (el valor de la función *HIGH* en *digitalWrite* significa 5V)
- Esperar a 1000ms (usando la función de *delay*)
- Coloque 0V ('LOW') en el puerto 13
- Espere 1000ms más.

En este ejemplo se utiliza el LED ya instalado en el puerto 13 de la placa Arduino, pero podemos usar un LED externo, o de cualquier otro circuito que "entienda" las señales de salida de la Arduino (0V y 5V), por lo tanto no sólo puede controlarse LEDs, sino también: motores, *drivers de relays* para el control de electrodomésticos y muchas otras cosas.

Todo lo que está en la función *loop* se repite continuamente (hasta desconectar el Arduino de la energía), por lo que el LED se enciende y apaga continuamente (1 segundo encedido y 1 segundo apagado).

Comunicación con Python

El primer ejemplo muestra una aplicación *stand alone* Arduino, es decir, software que se ejecuta en el Arduino y que no necesita una computadora para trabajar. Pero podemos crear una comunicación entre el dispositivo y la computadora, entonces un software en la PC (escrito en Python, por ejemplo) puede tomar decisiones más elaboradas, consultar bases de datos, sitios web y otros recursos y después, enviar un comando al dispositivo. Para esto utilizamos la librería *Serial*:

```
#define LED 13

void setup() {
    pinMode(LED, OUTPUT);
    Serial.begin(9600);
}

void loop() {
    if (Serial.available()) {
        char c = Serial.read();
        if (c == 'H') {
            digitalWrite(LED, HIGH);
        }
        else if (c == 'L') {
            digitalWrite(LED, LOW);
        }
    }
}
```

Con el código anterior, el Arduino enciende el LED cuando se recibe el carácter *H* a través de su puerto serie y apaga el LED cuando recibe *L*.

Cabe aclarar que Arduino tiene un convertidor USB-serie, por lo que podemos enviar a estos carácter a través de USB, usando el software en Python; y es así como tenemos un código en Python para controlar Arduino.

Antes de escribir el código en Python, es necesario instalar la librería *PySerial* [17]. Si usas Debian o Ubuntu, podrías usar el siguiente comando:

::

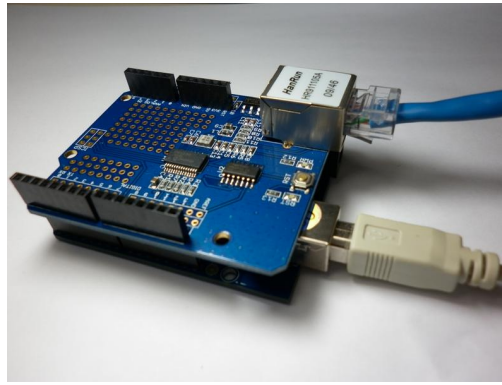
sudo aptitude install python-serial

Una vez instalada la biblioteca, vamos a crear un código de Python para controlar el LED:

```
#!/usr/bin/env python

import serial
import time

# /dev/ttyACM0 = Arduino Uno on Linux
# /dev/ttyUSB0 = Arduino Duemilanove on Linux

arduino = serial.Serial('/dev/ttyACM0', 9600)
time.sleep(2) # waiting the initialization...

arduino.write('H') # turns LED ON
time.sleep(1) # waits for 1 second

arduino.write('L') # turns LED OFF
time.sleep(1) # waits for 1 s   :height: 100px
:width: 200 pxecond

arduino.close() #say goodbye to Arduino
```

¡Listo! Ahora podemos controlar el LED de la placa Arduino con nuestro software en Python. Tambien, como ya se dijo, podemos basar la decisión (de activar o desactivar) en informaciónes externas como contenido de una tabla en una base de datos o los datos devueltos de una API de algún software o sitio web (como Twitter, Facebook, etc.).

En el ejemplo anterior, se utiliza un cable USB para la comunicación entre Python y el software de Arduino; pero podríamos haber conectado en el a la placa un *shield*, que es una componente que le da una nueva función como puede ser la comunicación vía Ethernet, Bluetooth o el control de un motor.



Proyectos

Actualmente trabajo en la enseñanza, dando cursos de Arduino en Brasil. Así que he desarrollado varios proyectos, especializados en la integración de Arduino con Python. Éstos son algunos de ellos:

- **Guitarrino**

  Mi novia, Tatiana Al-Chueyr, tuvo la idea de construir una guitarra para jugar Frets on Fire [18] (clon libre de Guitar hero, escrito en Python utilizando Pygame).

  El proyecto consiste en un simple circuito electrónico para botones, Arduino, un módulo Bluetooth y un software escrito en Python para recibir los datos (utilizando la librería pyBluez [19]) y hacer la integración con el juego.

  Más información en: http://github.com/tatiana/guitarrino

- **Droiduino**

  ¡Controle de cualquier dispositivo en su casa con un teléfono Android, a través de la Internet!

  Más información en: http://github.com/turicas/droiduino

- **Tweetlamp**

  Este es un ejemplo sencillo de cómo utilizar la PySerial para controlar una lámpara AC a través de Twitter.

  Más información en: http://github.com/turicas/tweetlamp

- **Turiquinhas**

  Controle con wifi un pequeño automovil con mando a distancia, utilizando por Wi-Fi y una interfaz Web.

  Más información en: http://justen.eng.br/turiquinhas

Si te quedaste con ganas de más, podés visitar la página oficial de arudino en http://arduino.cc/ o contactarte con el autor.

- **GitHub:** http://github.com/turicas
- **Cursos de Arduino:** http://www.CursoDeArduino.com.br

17 http://pyserial.sourceforge.net/

18 http://fretsonfire.sourceforge.net/

19 http://code.google.com/p/pybluez/

Lib Free or Die Hard

Autor: Juan B. Cabral

Bio: JBC conoció Python una solitaria noche del 2007. Desarrolló su proyecto de grado de la carrera Ingeniería en Sistemas con este lenguaje utilizando el framework Django y trabajó 1 año desarrollando evaluadores de información usando nuestro querido reptil.

Web: http://jbcabral.wordpress.com

Email: jbc.develop@gmail.com

Twitter: @juanbcabral

Todos sabemos que las librerías son fragmentos de código que **usamos para resolver problemas comunes**, las cuales podemos asumir como una caja negra que no sabemos **cómo** hacen su tarea, pero si **que** es lo que hacen. Podríamos decir que una librería nos da un poco de **desconocimiento** (**clueless** en ingles) sobre la solución de un problema.

Si bien este artículo no versa exclusivamente sobre prácticas para aplicarse a Python y puede aplicarse a toda la variedad de lenguajes de programación hay varios ejemplos que estan atados a la sintaxis del lenguaje de la víbora.

Muchos de los consejos y ideas que se exponen pueden encontrarse ejemplificados con Java en el libro “Practical API Design” [20] , el cual sugiero como lectura si se desea interiorisarse en el diseño de APIs.

Definiciones Previas

A las funcionalidades de una librería las accedemos a través de las API, la cual es la descripción de la librería, y es lo que llamamos para alcanzar el objetivo por el cual decidimos incorportar la librería a nuestro proyecto. Una buena API tiene un **“correcto nivel”** de **“clueless”** (la clave es que le nivel tiene que ser **correcto**).

Como dato anecdótico un subtipo de APIs son los **SPI**. Los cuales son funcionalidades que hay que extender o implementar para alcanzar un objetivo. Son la interfaz de un programa con un plugin. Un ejemplo de un **SPE** son los métodos que nos obliga a definir una clase abstracta.

```python
import abc

# ABCMeta es parte del API del modulo abc
class AbstractClass(object):

    __metaclass__ = abc.ABCMeta

    # Esta funcion es publica y se expone como parte del API de una
    # instancia de AbstractClass
    def public_function(self):
        pass

    # Esta funcion es publica y se expone como parte del API de una
    # instancia de AbstractClass
    def another_public_function(self):
        pass

    # Esta funcion NO es publica y NO es parte parte del API de una
    # instancia de AbstractClass
    def _private_function(self):
        pass

    # Esta funcion es publica y obliga a su redefinicion en una subclase
    # de AbstractClass por lo cual corresponde al SPE de la clase
    @abc.abstractmethod
    def abs_method(self):
        pass
```

Clueless

No Clueless (http://xkcd.com/676/)

El chiste anterior muestra de una manera irónica y cínica que sucede cuando vemos una solución informática sin un correcto nivel de clueless, pasando por cada nivel de abstracción disponible en el sistema.

Así el principio de **clueless** se basa en un idea muy simples: **La ignorancia es un beneficio**; la cual nos ayuda a enforcarnos en el problema a solucionar y no nos dispersa en **rediseños de ruedas**, que en general ya cuentan con soluciones bien realizadas en librerías ya existentes que vamos a usar sin que nos importe conocer su funcionamiento interno.

Cabe aclarar que es una idea que para quedarse (y es en cierta medida lo que provee las **n** capas de abstracción sobre el hardware que hoy tenemos), y que **NO** significa "no saber", sino, no preocuparse en lo que es el **core** de de tu programa y no funcionalidades ya existentes.

Como último detalle aclaramos que Python es altamente "clueless" ya que nos evita pensar en cosas de bajo nivel que otros lenguajes no.

Zen Vs. Zen

Las librerías al menos contradicen de alguna manera el "zen" de Python en los siguientes puntos:

- **Explicit is better than implicit:** Ya que ocultamos cierta funcionalidad adentro de un tipo de "caja negra".
- **Flat is better than nested:** La funcionalidad se encuentra anidada en la "caja negra" antes mencionada.

- **Special cases aren't special enough to break the rules:** Si consideramos a Python pelado como la forma "común" de hacer las cosas, utilizar una solución de terceros sería no utilizar la forma común, entonces estaríamos frente a un caso especial donde no utilizamos las reglas comunes de solución de problemas.
- **There should be one— and preferably only one —obvious way to do it:** Muchas librerías que resuelven un mismo problema existen (django, web2py, bottle, etc) por lo cual no hay una sola forma evidente de hacer las cosas.

¿Por qué es importante recordar estos puntos?, sencillamente porque ayudan a tomar decisiones de qué librería usar para cada problema, es importante asumir que cuando nos decidimos a utilizar una librería (o diseñarla) estamos atando nuestro curso de trabajo futuro a una forma **no estándar**, que es lo que intenta prevernirnos el Zen de Python. Así que pensar en utilizar o diseñar una librería es un punto bastante crítico a tener en cuenta; así como también es importante recordar estos (otros) aformismos también presentes en el Zen que evitan que caigamos en:

- **Although practicality beats purity:** Es mucho mas práctico utilizar una librería que empezar a escribir todo desde cero.
- **Namespaces are one honking great idea — let's do more of those!** Las librerías son los espacios de nombres mas cómodos de utilizar ya que se mantienen totalmente ajenos a nuestro desarollo

Así que dado que hay cosas contradictorias en las decisiones de diseños de API's y librerías voy a continuar dando unos consejos útiles a la hora de tomar la decisión de encarar un proyecto de este tipo para que se ajuste lo mejor posible a los principios del "estandaridad" de uso que plantea el Zen de Python.

Consejos

Exponer solo los métodos necesarios: Tratar de exponer muchos métodos, clases, funciones y variables públicos puede confundir al usuario de tu API. Tratar de mantener las llamadas públicas justas, que sirvan para dar la funcionalidad de la librería y no mucho mas. Tampoco hay que excederse con esto ya que si dejamos muy rígida nuestra librería, por la falta de comportamiento público puede que en algunos casos se vuelva inútil. En definitiva, debería cumplirse esto:

```
>>> len([n for n in dir(obj) if not n.startswith("_")])
(numero pequeño)
```

No exponer jerarquías profundas: No es lo mismo diseñar para la API que para reusar código. Exponer una gran cantidad de clases anidadas no suele ser útil al momento de solucionar un problema sin interesarnos en “cómo” se soluciona.

Cooperación con otras APIs: Nuestras librerías no están solas en el mundo, eventualmente se van a conectar con otras librerías de terceros y desde su concepción están conectadas, casi con seguridad, con la standar library de Python. Por estos motivos es que es importante mantener cierta “estandaridad” para lograr el Principio de menor sorpresa [21] en los usuarios. Para lograr esto es recomendable:

- Seguir la PEP 8 [2].
- Si usamos librerías de terceros evitar retornar objetos de la misma, excepto que nuestra librería necesite evidenciar ese acoplamiento (una app en Django seria un buen ejemplo).
- Ser cuidadoso en no redefinir comportamiento de otras APIs ya que aumenta el acoplamiento.

Mis tipos, tus tipos: Es mala idea exponer objetos de uso interno en las API, si esto es realmente necesario es buena idea que estos objetos tengan comportamiento muy similar a los de la biblioteca estándar. Así, si usamos algún tipo de colección iterable ordenada internamente, estaría bueno que se comporte como cualquier lista o tupla de Python (los query sets de Django cumplen esta idea).

Este principio también aplica a los formatos “estándar” de transferencia de datos

- **XML:** Trabajar con xml implica conocer muchas más cosas que la sintaxis del lenguaje; como ser: W3C DOM, XPath, XSL y otras apis que andan dando vuelta por ahi. Moraleja: XML debería tratar de no utilizarse.
- **JSON/YAML:** Mapean directamente sobre objetos del lenguaje que estemos utilizando (Python para nuestro caso) y no necesitamos más que conocer el manejo tradicional de listas y diccionarios para utilizarlos (y en el caso de YAML da un poco más de flexibilidad) lo cual hace que estos formatos posean un api más consistente con nuestra aplicación.

Controles de Tipos: Si bien Python posee Duck Typing los tipos de datos no esperados pueden causar serios dolores de cabeza en las librerías si no se validan lo antes posible. Por poner un ejemplo sencillo:

```
>>> def foo(a, b):
        c = a + b
        d = c * 2
```

```
        return d / 5

>>> print foo(1, 2) # [1]
1

>>> print foo("hello", "world") # [2]
Traceback (most recent call last):
  File "ej.py", line 9, in <module>
    print foo("hello", "world")
  File "ej.py", line 4, in foo
    return d / 5
TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

En el código anterior se ve como si bien la llamada a la funcion `foo` en **[1]** funciona a la perfección, el error que se muestra al ejecutar **[2]** realmente no radica (conceptualmente hablando) en la línea que dice `return d / 5` sino en que `foo` jamás espero que un string llegue como argumento. Dado que los controles de tipos llevan tiempo, utilizar la sentencia **assert** es buena idea para este trabajo, ya que pasando los test correspondientes con solo correr Python en modo optimizado en ambiente de producción (`$ python -O`) estas líneas no se ejecutaran. Nuestro ejemplo anterior entonces quedaría algo como:

```
>>> def foo(a, b):
    assert (a, (int, float))
    assert (b, (int, float))
    c = a + b
    d = c * 2
    return d / 5
```

Una última precaución a tener en cuenta es ser cuidadoso con los valores por **defecto** que se le da a las variables.

Errores Llamamos errores a algo inmanejable por nuestra librería y no a algo que Python considera un error. Así si python al tratar de abrir un archivo falla con un `IOError` puede que nuestra librería genere valores por defecto y continúe operando normalmente con lo cual dado nuestro dominio no implicaría un error. Se debe tratar los errores lo más tempranamente posible, totalmente lo opuesto a lo que sucede con la validación de tipos que se hacen a nivel de API, siempre teniendo en cuenta que **Errors should never pass silently, Unless explicitly silenced.**

Crear excepciones propias puede ser un arma de doble filo para nuestra librería ya que aumenta la capacidad de manejar errores desde la aplicación cliente pero Disminuye la homogeneidad con las **pilas**.

Inmutabilidad Rulz! Dado que a esta altura dijimos que tenemos que controlar todos los parámetros que llegan, todos los valores de retorno y todos los errores; Si llegamos a la necesidad de exponer objetos nuestos en nuestra API, al definirlos es buena idea intentar que sean inmutables.

Hacer objetos inmutables de alguna manera es darle todos los derechos de modificación al contructor (`__init__`) por lo cual toda validación que se necesite hacer de parámetros de entrada, esta concentrada en un solo lugar con lo cual ahorramos muchos dolores de cabeza.

Ya decidido si un objeto nuesto va a ser mutable o inmutable solo queda aclarar unas pequeñas cosas:

- **Si un objeto es inmutable:**

  -TRATAR de redefinir `__repr__`, `__str__`, `__hash__`,

  `__cmp__`, `__eq__` y `__ne__`.

- **Si un objeto es mutable:**

  - Controlar mucho lo que llega por las API.
  - Redefinir: `__repr__`, `__str__`, `__cmp__`, `__eq__` y `__ne__`.

Cuetiones de Diseño: Siempre planeen primero la funcionalidad o lo que es lo mismo decir: Primero el controller en MVC. O lo lo que es lo mismo **TDD**. Si bien en nuestra vida cotidiana es muy común que decidamos comprar nuestras cosas en función de una necesidad puntual, como programadores casi siempre cometemos el error de plantear primero “la cosa” y luego tratar de encontrarle una utilidad a lo que programamos. Es mucho más fácil y sencillo primero plantear un test. y al primer intento de correrlo evidentemente va a fallar por que estamos testeando algo que no construimos. Cuando dejamos de desarrollar… cuando el test pasa (acompaña esta revista una larga nota sobre diseño de MVC sobre las ideas de Maurice Merleau-Ponty el cual justifica de una manera filosófica el por que del TDD).

Sumado al TDD es también buena idea plantear inicialmente el nivel de excelencia que se quiere llegar nuestro proyecto y no simplemente liberar la primer versión cuando uno se aburre.

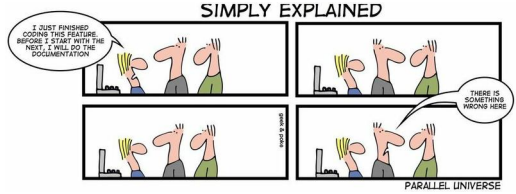
Portando Esencialmente hay dos formas de encarar el port de una librería en otro lenguaje a Python:

- **Facilitar a la vida a los desarrolladores Python con lo cual hay que**
    - Respetar pep8: `assertTrue -> assert_true/ asserttrue`
    - Utilizar funciones de Python: `obj.length() -> len(obj)`
    - Utilizar métodos de Python: `obj.appendChild(aobj) -> obj.append(aobj)`
- Facilitar a los usuarios que vengan del lenguaje de la librería original y quieran la misma funcionalidad en python, con lo cual lo ubico que hay que decir es que **Python no es Java** [22]

No publiquen sus librerías sin tests: Nada garantiza que lo que ustedes hicieron funcione en otros ambientes. Integrar los test a herramientas como setuptools es muy sencillo, y evita dolores de cabeza.

Publiquen sus librerías de manera comunes a los developers Python Si bien es bueno tener paquetes para todos los OS, prefieran cosas como PyPi que sus principales usuarios van a ser los propios desarrolladores.

No publiquen sus API sin documentación: Algo sin documentación es algo que no es usable, o en el peor de los caso obliga a sus usuarios a releer su código con lo cual quitan lo mas importante de un API (el cluelees). La documentación no sólo tiene que estar completa, sino de preferencia integrada y en el idioma de Shakespeare (aunque sea un mal Inglés).



Documentación
(http://geekandpoke.typepad.com/geekandpoke/2011/09/simply-explained-2.html)

Las APIs simétricas son buena idea: Una api simétrica es una que puede hacer algo en los dos sentidos, por ejemplo: Parsear JSON y generar JSON. Que una api sea simétrica aumenta mucho la estabilidad de un proyecto ya que resulta mucho mas simple testearlo. Por ejemplo una sencilla prueba de PyYaml sería algo como:

```
class TestA(unittest.TestCase):
   "el test es super sencillo y muy trivial"
```

```
    def setUp(self):
        # creamos un diccionario random
        self.data = dict((k, str(random.randint(1, 10000)))
                         for k in range(1000))

    def test_y(self):
        #testeamos el dump y el load de yaml
        self.assertEquals(yaml.load(yaml.dump(self.data)), self.data)

unittest.main()
```

La retrocompatibilidad es un compromiso: Si bien la mantención de código viejo puede ser tedioso y desde algunos puntos de vista inútil; al hacer algo público en nuestra API, estamos firmando un contrato con quien utiliza nuestra librería; con lo cual la decisión de eliminar código viejo público no es algo que debe hacerse a la ligera y es buena práctica deprecar lo que este pensando en eliminarse en versiones con anterioridad a la purga.

Conclusión

Todo buen diseño no deja de ser una cuestión subjetiva y puede que muchos de los lectores de esta nota no estén de acuerdo con cosas que aquí expongo, pero muchas de estas notas pueden ser buenos puntos de partida y mientras se evite el monkeypatch y no se abuse de los patrones, lo demás queda a juicio del cada alma libre.

20 Jaroslav Tulach, **“Practical API Design: Confessions of a Java Framework Architect”**, Apress; 1 edition (July 29, 2008), ISBN-10: 1430209739, ISBN-13: 978-1430209737

21 http://en.wikipedia.org/wiki/Principle_of_least_astonishment

22 http://dirtsimple.org/2004/12/python-is-not-java.html

from gc import commonsense - Optimización de memoria

Autor: Claudio Freire

Bio: Python, C, C++, Java, assembler, código máquina, lo que quieras.

Email: freireclaudio@yahoo.com.ar

Estuvimos hablando en esta serie primordialmente del colector de basura, el componente de Python que administra la memoria automáticamente. Comentamos cómo funciona, sus limitaciones, cómo evitar goteras de memoria y fragmentación.

Ahora cambiaremos un poco punto de vista, para enfocarnos en la eficiencia. Analizaremos cuánto ocupan en memoria los objetos Python, técnicas y trucos para hacer el uso más eficiente de la misma.

Objetos: PyObject

Todos los objetos en Python están representados por una estructura que incluye ciertos datos que el intérprete necesita para rastrear sus valores; algunos de ellos son: *contador de referencias, el tipo de datos, y algunos otros*. Aunque no todos los objetos almacenan la misma información adicional, por ejemplo: los que no son capaces de generar ciclos no necesitarán ser rastreados por el colector de ciclos.

El siguiente fragmento de código es el encabezado mínimo de los objetos en Python: Un contador de referencias, cuyo tamaño depende de la definición de `Py_ssize_t` (ya entraremos en este detalle), y un puntero al tipo de datos (que es otro `PyObject` particular, un `PyTypeObject`, que es la estructura de donde “heredan” todos los tipos).

```
Py_ssize_t ob_refcnt;
PyTypeObject *ob_type;
```

Cuando los objetos deben formar parte del colector de ciclos el encabezado queda como el siguiente.

```
/* GC information is stored BEFORE the object structure. */
typedef union _gc_head {
    struct {
        union _gc_head *gc_next;
        union _gc_head *gc_prev;
        Py_ssize_t gc_refs;
    } gc;
    long double dummy;  /* force worst-case alignment */
} PyGC_Head;
```

Nótese que ésto incluye a todas las instancias de clases definidas por el usuario, y gran cantidad de clases de extensión. De hecho, muy pocos tipos de objetos no necesitan este encabezado, notablemente entre ellos los tipos numéricos y las cadenas.

Para aquellos que no estén acostumbrados a leer C, esta estructura define dos punteros y un contador de referencias (adicional al del encabezado primordial que vimos antes). Con "dummy" se fuerza al tamaño de la estructura a ser, al menos, tan grande como un número de coma flotante de doble precisión, buscando con esto que el resto de los datos (el encabezado primordial), estén alineados lo mejor posible (puesto que esto afecta mucho a la performance de un programa).

También hay otra variante de objetos: *los de tamaño variable*. Estos objetos proveen un campo extra en su encabezado, para que el intérprete sepa que tamaño tienen.

32 vs. 64 bits

Para poner esto en perspectiva, necesitamos saber cuánto ocupan los punteros y los contadores. Estos tipos de datos no emplean los mismos recursos en todas las arquitecturas. Como su nombre lo indica, las arquitecturas de 32 bits tendrán punteros de 32 bits, y las de 64 bits, punteros de 64 bits.

Los números que representan tamaños (usualmente `long`), también tienen 32 y 64 bits respectivamente. Esto incluye al contador de referencias (`Py_ssize_t`), y muchos otros tipos primitivos dado que en arquitecturas de 64 bits, es ese el tamaño de los registros de hardware con los que opera el procesador.

Tipos primitivos

Entre los tipos primitivos de Python, tenemos la siguiente clasificación:

- Sin GC, tamaño fijo: `int`, `long`, `bool`, `float`

- Sin GC, tamaño variable: `str`
- Con GC, tamaño variable: `tuple`
- Con GC, tamaño fijo: `list`, `set`, `dict`, `unicode`, instancias de tipos del usuario, casi todo el resto.

Referencias: PyObject

Cada vez que se guarda una referencia a un objeto, Python lo que guarda es un puntero al PyObject.

Las tuplas son, pues, un array de punteros. Las listas, también, aunque de tamaño variable. Cada variable de nuestro código, cada atributo de una clase, cada parámetro pasado a cada función y presente en la pila, va a ocupar, pues, como mínimo, lo que ocupa un puntero. 64 o 32 bits.

Es claro que muchas veces ocuparán mucho más. Un atributo que esté guardado en el `__dict__` de la clase, ocupará una entrada en el diccionario, que incluye un puntero para el valor, otro para la clave (el nombre del atributo), y el hash de la clave. Este costo se paga por cada variable de cada instancia, lo que se hace significativo a la hora de evaluar el consumo de memoria.

Con esto, podemos armar la siguiente tabla para cada arquitectura

tipo	32bits	64 bits
PyObject	8	16
GC + PyObject	20	40
bool	12	24
int	12	24
long	16+bits	28+bits
float	16	24
str	20+len	36+len
unicode	24+4*len	48+4*len
tuple	24+4*len	48+8*len
list	32+4*len	64+8*len
set	48+64+8*len	96+128+16*len

dict	40+96+12*len	80+192+24*len
instancia	32+dict	64+dict

Slots

Teniendo en cuenta la tabla anterior, se puede notar que guardar datos masivos en estructuras hermosamente encapsuladas (objetos con atributos) puede llegar a ser un desperdicio de memoria. En particular, cuando los datos en sí son pequeños, números, bools, cadenas pequeñas, donde el overhead de las estructuras es mucho mayor al contenido en sí.

Sin renunciar al encapsulamiento, hay una forma de reducir este overhead. Es posible, en Python, prescindir del diccionario que almacena los atributos, almacenándolos directamente en el `PyObject` y, de esta forma, reduciendo a un tercio o incluso menos el costo de referenciar un valor dentro de un atributo.

No es una práctica de la que convenga abusar, sin embargo. Una clase que utilice este mecanismo se comportará notoriamente diferente de una clase regular, no se podrá inyectar atributos arbitrarios a la clase, no se podrá crear referencias débiles a sus instancias a menos que se haya tenido esa previsión (ya veremos cómo), y las cosas se complican bastante con respecto a las subclases.

Pero ciertamente es un mecanismo muy útil cuando queremos guardar “registros” de datos de forma prolija y *pythónica*, sin despilfarrar memoria.

```
class Registro(object)
    __slots__ = ( 'id', 'nombre', 'dato1', 'dato2' )

    def __init__(self, id, nombre, dato1, dato2):
        self.id = id
        self.nombre = nombre
        self.dato1 = dato1
        self.dato2 = dato2
```

Esta clase “Registro” ocupará, exactamente, `GC + PyObject + 4 punteros`, mucho más eficiente que una clase normal, igual de eficiente que las tuplas con nombre. No soporta referencias débiles ni atributos arbitrarios.

Este mecanismo sólo funciona con clases nuevas (que heredan de `object`), si se intentara usar sobre clases viejas (que no heredan de `object`), no tendrá efecto.

Si queremos referencias débiles, se deberá incluir `__weakref__` a los `__slots__`, y si queremos atributos arbitrarios habrá de agregarse `__dict__` a los mismos, aunque así habremos perdido mucha de las ventajas de utilizar `__slots__`.

El acceso a través de `slots` es apenas más lento que usando una clase regular, lo cual es algo anti-intuitivo dado que pareciera más rápido. Pero Python debe asociar el nombre del atributo a un offset dentro del `PyObject`, lo cual se hace de manera menos eficiente que a través de los altamente optimizados diccionarios de Python.

La pérdida de eficiencia en tiempo de acceso bien vale el ahorro de memoria, sin embargo, y cuando los datos son masivos, puede que incluso resulte en un programa más rápido, puesto que deberá barajar menos memoria para realizar su trabajo.

Otra ventaja es que ahora el objeto está contenido en una estructura compacta, sin la indirección del diccionario. Esto es una gran mejora, pues reduce notoriamente la fragmentación de memoria y, nuevamente, puede acelerar el programa con datos masivos.

```
class SubRegistro(Registro):
    pass
```

Al contrario de lo que pueda parecer, la clase `SubRegistro` sí tiene un diccionario. Para mantener las ventajas de los `slots`, es necesario especificarlos (aunque sea una lista vacía) en las subclases, o volverán a utilizar el diccionario como en las clases regulares. Para que esto no suceda:

```
class SubRegistro(Registro):
    __slots__ = ()
```

Esta clase heredará los `slots` de `Registro`, aunque no estén especificados. Si se agregaran `slots` a una clase base que no los utilice, la clase seguirá teniendo un diccionario, y no tendrá efecto.

Además, no funcionará la práctica usual de usar variables de clase como *default* de las correspondientes variables de instancia:

```
class Algo:
    verdadero = True

>>> a = Algo()
```

```
>>> a.verdadero
True
>>> a.verdadero = False
>>> a.verdadero
False
>>> Algo().verdadero
True

class Algo(object):
    __slots__ = ('verdadero',)
    verdadero = True

>>> a = Algo()
>>> a.verdadero
True
>>> a.verdadero = False
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'Algo' object attribute 'verdadero' is read-only
```

str vs. unicode

Otra interesante perla de la tabla anterior es que, aunque que los objetos unicode son muy buenos al tratar con texto internacional, despilfarran mucha memoria (4 bytes por caracter)

Python puede compilarse para que esto sean 2 bytes por caracter, usando UCS-2 en vez de UCS-4 como encoding interno, pero las distribuciones que he encontrado todas usan UCS-4.

De cualquier manera, hay una forma mucho más eficiente de almacenar cadenas. Y, si vamos a almacenar una gran cantidad, conviene tenerlo en cuenta: UTF-8.

UTF-8 es un encoding que soporta todo lo que UCS-4 soporta, pero muchísimo más eficientemente. El problema con UTF-8 es que no es para nada benigno al momento de manipular estas cadenas, obligando a transformarlas en unicode antes de aplicar cualquier tipo de manipulación sobre ellas.

De cualquier manera, en general, UTF-8 provee una eficiencia cercana al byte por caracter, cuatro veces más eficiente que UCS-4. Por lo tanto, cuando se necesiten grandes cantidades de cadenas, conviene usar UTF-8 en vez de meramente unicode,

sencillamente aplicando `x.encode("utf-8")` al guardar, y `x.decode("utf-8")` antes de manipular.

Deduplicación

Otro truco a tener en cuenta al manipular cadenas, es la deduplicación de las mismas. Es bastante común, al generar grandes estructuras de datos con cadenas en ellas, olvidarse del pequeño detalle de que dos `"abracadabra"` pueden no ocupar el mismo espacio de memoria.

Si tengo un archivo con dos líneas `abracadabra`, hacer:

```
l = open(archivo,"r").readlines()
```

No va a darme una lista con dos referencias a un string `"abracadabra"`, sino una lista con dos strings, que de casualidad contienen el mismo contenido.

Al procesar datos de esta manera, conviene deduplicar las cadenas, usando lo que se conoce como un *"mapa de identidad"*:

```
idmap = {}
lineas = []
for l in open(archivo,"r"):
    lineas.append(idmap.setdefault(l,l))
```

Al pasarlo por el mapa de identidad, sólo una copia de cada cadena se mantendrá en memoria, y, dependiendo de los datos, claro está, podríamos ahorrarnos gran cantidad de memoria.

Esto es perfectamente seguro puesto que las cadenas son objetos inmutables. De hecho, Python provee una función que hace esto mismo, `intern`, aunque a veces conviene evitar su uso: Las cadenas `intern` adas se introducen en un mapa de identidades interno de Python que se utiliza para el acceso a atributos y otros lugares donde hace falta alta performance. Abusar del mismo puede resultar en una performance seriamente degradada en lugares inesperados.

struct

Alternativamente a los slots, una manera interesante (aunque poco elegante) de empaquetar información, es en cadenas. Las ventajas, en cuanto a uso de memoria, son considerables, puesto que las cadenas son simplemente *bytes* para Python.

Si almacenamos estructuras grandes en una cadena, reduciremos el overhead impuesto por Python a casi nada, aunque lo pagaremos al momento de acceder a la información, tanto en prolijidad perdida del código como overhead al interpretar la cadena con Python.

El módulo `struct` y `mmap` nos puede ayudar mucho con esto. Imaginemos, pues, que tenemos un registro que son dos enteros de 32 bits más una cadena de 12 caracteres. En total, ocupará 20 bytes. Imaginemos, además, que queremos todo un array de estas cosas, 100 digamos. Ocupará, a lo mejor, 2000 bytes.

```
>>> formato = 'ii 12s'
>>> longitud = len(struct.pack(formato, 0, 0, ''))
>>> buffer = mmap.mmap(-1, longitud*100)
>>> buffer[0:longitud] = struct.pack(formato, 1, 2, 'hola')
>>> struct.unpack(formato, buffer[0:longitud])
(1, 2, 'hola\x00\x00\x00\x00\x00\x00\x00\x00')
```

Ahí se podemos ver un pequeño inconveniente: la forma en que se guarda la cadena (con bytes nulos al final), pero cuando trabajemos con datos masivos y altamente estructurados (como registros de bases de datos), puede que sea una opción a tener en cuenta. En particular, si no querremos acceder muy frecuentemente a los datos.

No es muy pythónico, pero la diferencia en uso de memoria es importante. El módulo `mmap` utiliza al sistema operativo para proveer un pedazo de memoria en el cual se puede escribir datos (de hecho, el pedazo de memoria puede mapearse a un archivo o compartir entre procesos). Pero es sólo eficiente si los datos van a ocupar más (mucho más) de 4kb, pues el sistema reserva una cantidad integral de **páginas** de memoria, no sólo bytes.

La técnica puede funcionar igualmente bien con cadenas, con el inconveniente de que las cadenas no son mutables, así que cambiar su contenido conlleva el costo de copiar la estructura entera.

Vectorización

Esta técnica se puede llevar a una forma mucho más eficiente y cómoda de utilizar aplicando una técnica conocida como *vectorización*.

Donde los registros de datos que usemos sean prominentemente numéricos, hay un módulo que nos permitirá guardar listas enteras de ellos de forma eficiente: `array`.

```
>>> pos_x = array.array('i')
>>> pos_y = array.array('i')
>>> for i in xrange(1000):
...     pos_x.append(i)
...     pos_y.append(i)
```

Eso creará dos arreglos de 1000 enteros, y ocupará exactamente lo que ocuparía ese mismo arreglo en C. Lo cual es muy eficiente. Y no es tan difícil de usar, puesto que `array` es idéntico a `list` en cuanto a su operación, con la diferencia de que sólo acepta un tipo predefinido de datos (*i* = integer, hay muchos otros), y los mantiene en memoria de forma realmente compacta y eficiente.

De hecho, se puede armar una clase que funcione de "vista":

```
class Point(object):
    __slots__ = ('_x','_y','_i')
    def __init__(x, y, i):
       self._x = x
       self._y = y
       self._i = i

    @property
    def x(self):
       return self._x[self._i]

    @x.setter(self, value):
       self._x[self._i] = value

    @property
    def y(self):
       return self._y[self._i]

    @y.setter(self, value):
       self._y[self._i] = value
```

Por supuesto, `Point` ocupa bastante memoria, pero sólo necesitamos estos objetos cuando manipulemos entradas dentro del array. Son sólo una vista para los datos del array, lo que los hace muy útiles. Además, dejamos el índice mutable, así que podemos hacer:

```
p = Point(pos_x, pos_y, 0)
for i in xrange(len(pos_x)):
   p._i = i
   # hacer algo con p
```

Y tendremos una manera bastante pythónica de manipular nuestros datos.

Esta técnica de *vistas* es igualmente aplicable a `struct` y muchas otras técnicas de compactación de datos. Conviene preparar un módulo y tenerlo a mano para esta tarea.

Buffers

Finalmente, tenemos que considerar el overhead de empaquetar cadenas pequeñas en una grande, como hicimos con `struct`. Cada vez que queremos leer un registro, hay que copiarlo (`buf[0:longitud]` crea una copia).

Una manera de evitar estas copias, es utilizando los *buffers* que provee Python. Son vistas de cadenas, una manera de hacer *slices* sin copiar los datos, que es parte del lenguaje.

Luego, si queremos armar una clase que funcione como un array de tamaño fijo de cadenas de tamaño fijo, usando *buffers* podemos evitarnos la copia de las cadenas a cada acceso:

```
class StringArray(object):

    def __init__(self, long_elemento, elementos):
        self._long_elemento = long_elemento
        self._buf = mmap.mmap(-1, elementos*long_elemento)

    def __len__(self):
        return len(self._buf) / self._long_elemento

    def __getitem__(self, index):
        return buffer(self._buf, index*self._long_elemento, self._long_elemento)

    def __setitem__(self, index, value):
        if index < 0:
            index += len(self)
```

```
        self._buf[index*self._long_elemento:(index+1)*self._long_elemento] = value

    def __iter__(self):
        for i in xrange(len(self)):
            yield self[i]
```

Como tal vez note en el código, los *buffers* son de sólo lectura, así que para escribir en el buffer aún necesitamos la notación de slices usual. Aún así, de esta forma podemos trabajar eficientemente con listas de cadenas de longitud fija, lo cual, como vimos, pueden encapsular estructuras de datos bastante variadas.

Casi todas las funciones de Python que aceptan cadenas también aceptan buffers, aunque no todas, así que podemos tratarlos casi como cadenas normales.

Compresión

Finalmente, tenemos la opción de usar todas estas técnicas, en conjunto con técnicas de compresión de datos.

Comprimir datos en memoria es un problema completamente diferente de la compresión de datos para almacenamiento. En memoria, necesitamos poder utilizar, y a veces modificar los datos. No es trivial el desarrollar un esquema de compresión que permita realizar de forma eficiente estas tareas.

Un par bastante común es, en indexación, las técnicas conocida como *delta-coding* y *prefix-coding*. Estas técnicas permiten guardar conjuntos de números (para delta) o cadenas (para prefix) de forma eficiente.

En ambos métodos se guardan conjuntos sin un orden particular. Es decir, se guardan los números *a*, *b* y *c*, pero no el orden en que se agregan al conjunto. Al no guardar el orden, la estructura se puede ordenar en memoria de forma tal que ocupe la menor cantidad de memoria posible.

Veremos el caso de estructuras inmutables, puesto que la mutabilidad llevaría todo un estudio que puede leerse de internet. Pero es posible implementar versiones de estas estructuras que aceptan modificarse, aunque de forma limitada.

En el caso de números, primero el conjunto se ordena, y luego se guarda cada número como la diferencia entre ese y el número previo. De esta manera, hace falta guardar números más pequeños, que es posible quepan en un byte, en vez de 4 u 8. Para poder aprovechar esto, los números se guardan en algo llamado *rice code*, que para números pequeños guarda un único byte, y para números más grandes guarda más bytes. Es lo que se llama un código de longitud variable.

Una clase CompressedSet se puede ver como:

```
class CompressedSet(object):

    def __init__(self, datos):
        self.datos = array.array('B')
        ini = 0
        for i in sorted(datos):
            delta = i - ini
            ini = i
            while 1:
                more = delta > 0x7f
                self.datos.append(0x80 * more + (delta & 0x7f))
                delta >>= 7
                if not more:
                    break

    def __contains__(self, valor):
        for i in self:
            if i == valor:
                return True
            elif i > valor:
                return False
        else:
            return False

    def __iter__(self):
        x = 0
        val = 0
        for i in self.datos:
            more = (i & 0x80) != 0
            val |= (i & 0x7f)
            if more:
                val <<= 7
            else:
                x += val
                val = 0
                yield x
```

Montones de optimizaciones se pueden hacer a esta clase. Montones de operaciones se pueden implementar sobre ella eficientemente. Por ejemplo, unión, intersección, resta y todas las operaciones de conjuntos se pueden implementar muy eficientemente. Pertenencia, `x in y`, no es tan eficiente sin embargo. Pero hay técnicas para mejorar esto considerablemente, sin perder la eficiencia en memoria, que es muy buena.

De hecho, hasta se puede comprimir más, si se aplica también RLE (compresión de corridas), aunque si vale la pena o no depende mucho de la aplicación.

Para el caso de *prefix-coding*, cada cadena es representada por un número, y por una cadena. El número indica cuán largo prefijo de la cadena anterior se puede usar, mientras que la cadena es qué hay que agregar al prefijo para obtener el valor final.

Al estar ordenadas las cadenas, la ocurrencia de prefijos es considerable. Esta técnica utiliza el comando *locate* de linux, y es particularmente útil cuando lo que se almacena son paths de archivos o urls, donde los prefijos son cosa común.

```
class CompressedStringSet(object):
    def __init__(self, datos):
        self.prefijos = array.array('I')
        self.sufijos = array.array('I')
        self.chars = array.array('c')

        anterior = ''
        for s in sorted(datos):
            long_prefijo = 0
            for c,c2 in zip(s,anterior):
                if c != c2:
                    break
                else:
                    long_prefijo += 1
            anterior = s
            self.prefijos.append(long_prefijo)
            self.sufijos.append(len(self.chars))
            self.sufijos.append(len(s) - long_prefijo)
            self.chars.extend(s[long_prefijo:])

    def __len__(self):
        return len(self.prefijos)
```

```
    def __iter__(self):
        anterior = ''
        for i in xrange(len(self)):
            long_prefijo = self.prefijos[i]
            pos_sufijo, long_sufijo = self.sufijos[i*2:i*2+2]
            anterior = anterior[:long_prefijo]+str(buffer(self.chars,pos_sufijo,long_sufijo))
            yield anterior

    def __getitem__(self, i):
        long_prefijo = self.prefijos[i]
        pos_sufijo, long_sufijo = self.sufijos[i*2:i*2+2]
        if long_prefijo:
            prefijo = self[i-1][:long_prefijo]
        else:
            prefijo = ''
        return prefijo+str(buffer(self.chars,pos_sufijo,long_sufijo))
```

Muchas cosas se pueden mejorar en esta estructura, pero ilustra el punto.

Conclusiones

Usar memoria eficientemente no es gratis. Hay que trabajar para lograrlo, y muchas veces en perjuicio de la simplicidad o elegancia del código. Pero en ocasiones es obligación tomarse el tiempo para hacerlo. Si la memoria escasea, usarla eficientemente es imperativo, no opcional.

En esos casos, con suerte, lo mencionado aquí ayudará a identificar los puntos débiles de nuestro código y las técnicas mencionadas una forma de mejorarlo.

Ciertamente las técnicas vistas no son todas las que existen, y claro que se puede decir muchísimo más del tema. Pero con lo que hemos tratado espero que el lector tenga ahora al menos una idea de dónde empezar a *googlear*.

Suerte, y hasta la próxima PET.

ORDENANDO MVC CON LAS IDEAS DE MERLEAU-PONTY

Autor: Javier Der Derian

Bio: Desarrollador Web desde hace unos 10 años, Freelance desde hace 4; y hace ya más de 1 año que empecé a formar un equipo de programadores para llevar mi proyecto freelance a algo más grande (una Empresa, digamos).

Python es mi herramienta de trabajo diaria y el lenguaje en el que más disfruto programar.

Hace ya más de 8 años que estudio Psicoanálisis y tengo el desafío de integrar en la práctica 2 cosas que parecen tan distantes como la programación y el psicoanálisis

Web: http://tribalo.net/

Email: javier@tribalo.net

Twitter: @JavyerDerDerian

Maurice Merleau-Ponty [23] fue un "Filósofo Fenomenólogo" Francés de principios del Siglo XX. En su obra más importante **"La Fenomenología de la Percepción"** [24] planteó, entre muchas otras cuestiones, un sistema con el cual ordenar lo que lo humanos percibimos.

Pero no es de lo que planteó Merleau-Ponty de lo que quiero hablarles, sino de lo que sobre sus trabajos desarrolló y planteó Jacques Lacan [25], Psicoanalista Francés que parado sobre los hombros de Freud [26], reencausó sus descubrimientos articulándolos con herramientas de la lingüística, topología y teoría de los juegos, entre otras.

Es articulando los desarrollos de Lacan sobre los 3 Registros con lo que planteo el ordenamiento del MVC [27].

Lacan + MVC + Merleau-Ponty

La Fenomenología de la Percepción

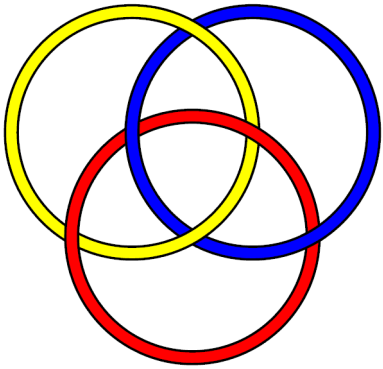
En “La Fenomenología de la Percepción” Merleau-Ponty plantea 3 Registros con los cuales ordenar la realidad:

- El Registro de lo Simbólico
- El Registro de lo Imaginario
- El Registro de lo Real

Lacan toma ese planteo y lo desarolla extensamente en su práctica.

¿Qué son los 3 Registros?

Son 3 Lazos que, anudados de distintas formas, ordenan la mente humana; y con eso la mente humana ordena lo que quiera ordenar; sea un edificio de 10 pisos o un software.



Nudo de Borromeo [28]

El distinto anudamiento de esos 3 Lazos produce tremendos caos o las creaciones más sublimes del humano. Todo depende del orden con el que se los anude.

Habiendo solamente un orden correcto, una forma eficaz, de hacer el nudo:

Simbólico -> Imaginario -> Real

Pero:

- ¿Qué me dice cada uno de esos nombres?
- ¿Qué es cada uno de esos registros?

Registro de lo Simbólico: Es el registro que hace al Nombre, al significante de las cuestiones; y es, además, el que ordena y determina sobre los otros dos.

Lo simbólico es, valga la redundancia, el registro con el cual una "cosa" es simbolizada y por lo tanto usable por el humano. Dependiendo del simbólico, del nombre, que le pongamos a "algo", el que ese "algo" tendrá funciones y posibilidades distintas.

Un ejemplo muy práctico de entender es el de una mesa de madera. Para la cultura occidental una mesa de madera es...una mesa de madera. Representa momentos de reunión familiar y se usa para comer. En otro tipo de cultura, en esa mesa verán y nombrarán a la madera. Y por lo tanto "eso" ya tendrá una función muy distinta: combustible. La misma "cosa" cambia en su funcionalidad totalmente por el Nombre con el que se la usa; y ese nombre es el que hace al registro simbólico.

Registro de lo Imaginario: Es el registro que, a través de las imágenes y las formas, une a un nombre con una "cosa". sin el registro imaginario, sin la imagen de una casa...la palabra casa puede referirse tanto a la casa de alguien, como al verbo "casar" realizado por un cura en el sacramento del matrimonio.

Es el Registro que recubre a las cosas, y es el que relaciona lo que decimos con la cosa a alcanzar.

Registro de lo Real: Es la "cosa" en sí; inalcanzable para el humano si no es a través de los otros registros.

Y, ahora que tenemos todo esto...¿Qué hacemos con MVC?

MVC es un paradigma de la programación que nos plantea 3 diferencias bien marcadas en cuanto a la programación.

Tenemos los **modelos**, las **vistas** y los **controladores**; cada uno con su lógica.

Pero MVC no plantea un orden, y por lo tanto lo que sucede es que, como en tantas cuestiones de la vida, lo que se termina construyendo es más un caos que lo que uno hubiera querido construir.

Tenemos 3 Registros y 3 sistemas a Integrar.

Empecemos por el que parece el más cercano.

El Registro de lo Real: "Eso" que si no fuera por los sistemas que usamos para accederlo nunca lograríamos entenderlo.

"Eso" que se guarda en una base de datos en formatos ilegibles para el humano; o, más lejos aún, en un disco rígido en algún remoto servidor en "la nube", como ceros y unos secuenciados que ningún humano entiende si no es a través de algún sistema que le permita leerlo.

Los modelos son el registro real; y no tendríamos acceso a los modelos, a la información que guardamos en ellos, si no fuera por todos los sistemas que nos permiten leerlos. La información en si misma es inutilizable; los ceros y unos; la variación magnética en un disco rígido a miles de kilómetros de mi casa es inaccesible; y hace a lo Real.

En el **registro de lo imaginario mapeamos a las Vistas**; la parte visible de nuestro software. La cual nos permite conectar a los controladores con la información que tenemos en los modelos. Y aunque cambie mil veces de forma (de colores, de estilos, de HTML a XHTML o a un PDF), la vista siempre nos hará de puente hacia los datos.

Por otra parte, en el **registro de lo simbólico tenemos a los controladores**; que con el mismo nombre ya nos muestra que es el lugar desde el cual se ordenará, controlará, todo lo que suceda en las otras partes del software.

La Propuesta es pensar la Programación desde este lugar que, aunque incómodo para los hábitos normales, nos propone una eficacia que normalmente no se alcanza.

Tomarse el trabajo de definir los controladores que llevará nuestra aplicación; una vez que tengamos esa base trabajada recién ahí pasar a definir las vistas, y por último y casi como consecuencia de todo lo anterior tendremos que desarrollar los modelos.

Por otro lado justifica el hecho de realizar TDD [29] ya que al diseñar los test primero nos obliga a definir las API's [30] lo cual obliga en cierta forma definir primero la funcionalidad (lo simbólico) antes que los datos.

23	http://en.wikipedia.org/wiki/Maurice_Merleau-Ponty
24	http://en.wikipedia.org/wiki/Phenomenology_of_Perception
25	http://en.wikipedia.org/wiki/Jacques_Lacan
26	http://en.wikipedia.org/wiki/Freud

27 http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

28 http://en.wikipedia.org/wiki/Borromean_rings

29 http://en.wikipedia.org/wiki/Test-driven_development

30 http://en.wikipedia.org/wiki/Api

Programando Cross-Platform: cosas a tener en cuenta para no quedarte pelado

Autor: Manuel de la Pena Saenz
Web: http://www.themacaque.com
Email: manuel.delapena@canonical.com
Twitter: @mandel_macaque

Cuando hablamos del termino cross-platfrom o multi-plataforma, en contexto de la informatica, nos referimos a aquel software o programa que es capaz de ejecutarse en más de un sistema operativo o plataforma. Con esta definición podemos decir que si nuestro software es capaz de ejecutarse en Windows sobre un x86, Unix x86 y Mac OS X PPC nuestro proyecto es multi-plataforma.

La gran mayoría de los programadores python sabemos que el interprete de nuestro lenguaje preferido es multi-plataforma, es decir, podemos tener un interprete de python en Windows x86, Ubuntu o Mac OS X que ejecutara nuestros scripts y aplicaciones. En problema con esta cualidad inherente de python, ser cross-platform, es que no es un a propiedad transitiva. El hecho de que nuestra aplicación este escrita sobre un lenguaje de programación que tiene un interprete multi-plataforma no implica que nuestra aplicación sea en si capaz de hacerlo.

1. Es realmente necesario?

A la hora de empezar un proyecto multi-plataforma hemos de siempre hacernos esta pregunta. Es realmente necesario que mi software soporte más de un sistema operativo?

El motivo de hacerse esta pregunta *“trascendental”* es el hecho de que hacer que una aplicación soporte más de una plataforma no es una tarea trivial. Aunque no hay números exactos sobre esto, por experiencia, se puede decir que cada vez que añadimos un nuevo sistema operativo a las plataformas que vamos a soportar, la complejidad de nuestro código crece. Esta complejidad es algo fundamental a tener en cuenta, ya que cuanto mayor sea, mayor serán los recursos que necesitemos, ya sea el numero de programadores, tiempo, inversión etc… Hay infinidad de motivos por los cuales nuestro software puede tener la necesidad de soportar más de una

sistema operativo, en ocasiones es un requerimiento de nuestro cliente, la necesidad de tener mayor audiencia o un *"feature"* que queremos proporcionar a nuestros usuarios.

2. Como abordar el diseño

A la hora de abordar un proyecto en el que queremos soportar más de un sistema hay diferentes diseños que se pueden utilizar.

2.1 Me sobra el dinero

A ciertas instituciones a la que el dinero parece crecerles en los árboles y son ellas las que se pueden permitir el lujo de crear una aplicación multi-plataforma a base de crear x versiones de código, donde x es el numero de plataformas que se van a soportar. Por supuesto esta opción tiene un gran número de costes añadidos:

- Ambos programas deben de ser parecidos, lo que no siempre es fácil.
- La complejidad a la hora de organizar equipos es mayor. Sea la de tener un organización que sea capaz de mantener los diferentes árboles de código.
- Es más difícil de seguir los diferentes bugs y fixes que se aplican al código.

2.2 El problema es de otro

Esta opción se refiere al caso en el que se ignoran tales diferencias ya que estamos trabajando en un framework que oculta las diferencias de las diferentes plataformas. Un de los mayores ejemplos de este diseño es Java, ya que nos garantiza que la API de Java siempre será la misma independientemente de la plataforma en la que se ejecute.

2.3 Puedo ser más inteligente

Es este caso estamos hablando de aquel desarrollo de software en el que la arquitectura ha tenido en cuenta aquel código que es agnóstico a la plataforma, es decir, aquel código que no tiene contacto directo con el sistema operativo. Lo más común en la mayoría de las aplicaciones, es que dicho código agnóstico contenga la mayoría de la lógica de nuestro proyecto, mientras que el código que ha de ser dependiente de la plataforma solo cóntega lógica referente a la plataforma que tiene como meta. En la mayoría de los casos, el código agnóstico es la mayor parte de la plataforma.

Las tres diferentes ideas explicadas son soluciones generales, pero tendiendo en cuenta que estamos trabajando en python, la opción obvia a elegir para un

programador python es la tercera. Pero cuidado, no porque podamos programar nuestro software de esa manera implica que hacer un programan multi-plataforma no tenga peligros para el desarrollador. Cuando una arquitectura se hace multi-plataforma podemos correr los siguientes riesgos:

- Los desarrolladores pueden estar formados a utilizar el mínimo común denominador de las aplicaciones.
- El testeo es mucho más y costoso, ya que se ha de realizar en cada plataforma. La automatización de la ejecución de los tests es mucho más laboriosa.
- Las diferentes plataformas tienen diferentes convenciones, por ejemplo, ¿dónde han de ir los botones de un dialogo en Mac OS X y en Ubuntu usando Gnome?

3. Como organizar mis módulos y paquetes

Una vez que hemos identificado los diferentes fragmentos de código que sabemos que no pueden ser agnósticos a la plataforma, debemos considerar la estructura que dicho código va a tener y cómo va a ser importado para poder utilizase desde el resto de nuestro proyecto.

3.1 Un paquete para gobernarlos a todos

La idea detrás de este diseño es crear un solo paquete que contenga todos aquellos fragmentos de código que son dependientes de la plataforma. El paquete puede contener diferentes módulos de forma que podamos organizar de una forma lógica todos los diferentes imports. Pongamos como ejemplo que el nombre de nuestra aplicación es myapp. En este diseño podemos hacer el import de la siguiente manera:

```
from myapp import platfrom

platform.open_file('file')
```

De esta forma podemos siempre asegurarnos que usaremos el código de la plataforma allá donde se necesite. La estructura del paquete puede ser la siguiente:

```
myapp/
    __init__.py
    platform/
         __init__.py
         linux/
```

```
            __init__.py
            io.py
            ipc.py
        windows/
            __init__.py
            io.py
            ipc.py
```

Siguiendo nuestra estructura de paquete anterior podemos imaginar que el __init__.py the windows y de linux pueden ser de la siguiente forma:

```
from myapp.platform.windows.io import open_file, make_readonly
from myapp.platform.windows.ipc import get_status, send_file
```

Nota: El código anterior es a modo de ejemplo.

Por supuesto mantener el __init__.py de los diferentes paquetes va a ser un trabajo extra, sobretodo si queremos que el paquete platform sea el que exporta las diferentes funciones, es decir queremos hacer el import como:

```
from myapp import platfrom
```

y no:

```
import sys
if sys.platfrom == 'win32':
    from myapp.platform import windows as platform
else:
    from myapp.platform import linux as platform
```

Para hacer el mantenimiento del __init__.py del paquete platform podemos utilizar el siguiente truco (obra del gran __lucio__):

```
import sys

# very hackish way to avoid "import *" to satisfy pyflakes
# and to avoid import myapp.platform.X as source (it wont work)
```

```
if sys.platform == "win32":
    from myapp.platform import windows
    source = windows
else:
    from myapp.platform import linux
    source = linux


target = sys.modules[__name__]
for k in dir(source):
    setattr(target, k, getattr(source, k))
```

Hasta aquí parece que esta solución es *óptima* ya que el código multiplataforma está divido y podemos importar todas aquellas funciones que necesitamos desde el mismo paquete. Por desgracia no es todo oro lo que reluce y esta forma de organizar el código tiene un número de problemas:

- Se ha de mantener los __init__.py de cada plataforma, esto puede resultar ser mucho trabajo.
- La arquitectura es mucho más frágil y propensa a *circular imports* que no es un problema fácil de solucionar.

3.2 Tres paquetes para los Reyes elfos bajo el cielo. Siete para los...

La idea detrás de este diseño es la de organizar nuestro código de forma lógica y no dejarnos influenciar por el hecho de que ciertas partes del código sean dependiente de la plataforma. Una vez que hemos hecho una división lógica, nos enfocamos en implementar nuestros módulos creando paquetes cuando se necesario, siguiente el ejemplo anterior tendríamos la siguiente organización (nessita es quizás la persona a la que agradecer esto):

```
myapp/
    __init__.py
    io/
        __init__.py
        windows.py
        linux.py
    ipc/
```

```
        __init__.py
        windows.py
        linux.py
```

Con este diseño pasamos de hacer:

```
from myapp.platform import open_file, send_status
```

a:

```
from myapp.io import open_file
from myapp.ipc import send_status
```

Esta solución es muy parecida a la forma en el que el paquete os funciona (aunque la implementación no es exactamente la misma). Este diseño reduce la posibilidad de que tengamos circular imports en el código (aunque no es una garantía) y ademas es mucho más elegante.

4. La interfaz de usuario

Una de las partes de un aplicación que está más expuesta al usuario es la interfaz de usuario. Cuando diseñemos la interfaz de usuario debemos de pensar en las diferentes formas de tomar el camino del desarrollo. Estas opciones pueden ser las siguientes:

- Crear una interfaz de usuario por plataforma de forma que la interfaz sea nativa al 100%.
- Utilizar un toolkit que intente minimizar las diferencias entre los diferentes OS.
- Crear una interfaz única que destaque y que no necesariamente sea nativa a la plataforma.

Está claro, que como en ocasiones anteriores, debemos de tomar el diseño que más nos convenga. Por ejemplo, si sabemos que nuestra aplicación va a tener un gran número de usuarios de Mac Os X debemos de recodar que suelen ser muy ‘quisquillosos’ con la interfaz de usuario, en especial, lo usuarios de Mac tienden a quejarse cuando se utiliza un toolkit que intenta imtar el estilo de Mac Os X (ya sea Qt, Gtk, WxWidgets, etc.) En este caso quizás sería recomendable intentar ir a por un diseño en el que se utiliza Cocoa.

Otra de las opciones que llama la atención es la de crear una GUI que no tenga casi parecido alguno con las interfaces nativas. Este camino lo suelen tomar muy a menudo los desarrolladores de juegos. La idea es simple, mi UI va a utilizar su propio estilo el cual va a ser constante en todas las plataformas que se soportan. Algunas de las ventajas de esta técnica son las siguientes:

- La interfaz de nuestra aplicación se va a destacar dentro del sistema del usuario. Si se realiza con gusto y tacto, este puede ser un gran punto fuerte de la aplicación.
- Al ser única, la interfaz no intenta parecer nativa por lo que es menos molesta. El ser humano ha evolucionado alrededor del sentido de la vista y puede notar la más minima diferencia. Intentar ocultar dichos errores puede ser más difícil que algunas partes del bussiness logic de la apliación. Haciendo una interfaz única nos ahorramos dicho trabajo.

Una vez elegido que camino tomar se ha de mirar que toolkit o framework nos interesa, por ejemplo, debemos de elegir entre:

- Qt (pyqt)
- FLTK (pyFLTK)
- FOX (PXPy)
- SDL
- SFML
- Allegro (pyallegro)
- Gtk+ (pygtk o gobject instrospection)
- MFC (windows)
- Tk

Listar las cualidades de cada una de las toolkits o poner un ejemplo es demasiado para este artículo (quizás otro solo sobre el tema sería interesante) así que dejo al lector con la inquietud :).

5. El file system, uno de nuestros enemigos

Uno de los puntos en lo que es más probable que tengamos problemas de multiplataforma, especialmente en Windows, es el file system. El stdlib de python con os intenta solucionar varios problemas de los que nos podemos econtrar pero no soluciona todos ellos.

5.1 Paths laaaaaaaargos

Cuando estamos interactuando con archivos hemos de tener encuenta cuales límites tiene el filesystm en el que trabajamos, una buena referencia es la página de la wikipedia que los compara (http://en.wikipedia.org/wiki/Comparison_of_file_systems). Una de las diferencias que hemos de tener en cuenta es el limite del path, por ejemplo si queremos soportar Windows y Linux tenemos que recordar los siguiente:

Sistema	Límite
ext2	Sin limite
ext3	Sin limite
ext4	Sin limite
NTFS	255 chars

En problema está claro, cómo hacemos para que nuestro usuario pueda usar de una forma similar nuestra aplicación en Windows y en Linux. Para esto tenemos dos opciones:

- Use el mínimo común, es decir, 255 caracteres
- Intentar usar un largo suficiente para que se poco común en nuestro use case.

La opción que elijamos depende de nuestro use case, en este articulo nos vamos a fijar como hacer que los paths en NTFS sean algo más largos. Para esto vamos a recurrir a utilizar literal paths en Windows:

```
The Windows API has many functions that also have Unicode versions to
permit an extended-length path for a maximum total path length of
32,767 characters. This type of path is composed of components
separated by backslashes, each up to the value returned in the
lpMaximumComponentLength parameter of the GetVolumeInformation
function (this value is commonly 255 characters). To specify an
extended-length path, use the "\\?\" prefix. For example,
"\\?\D:\very long path".
```

El anterior apartado quiere decir que la limitación de los 255 caracteres puede ser evitada usando un literal path, lo cual se hace usando el prefijo **\?**. Algo importante que se ha de saber sobre la API de Windows respecto a estos paths es lo siguiente: Todas la funciones de IO de Windows convierten “/” en “" excepto cuando **\?** es utilizado, esto lleva problemas en algunas de la funciones de os, por ejemplo

os.listdir, veamos un ejemplo: mientras lo siguiente es correcto:

```
import os
test = r'C:\Python7'
print os.listdir(test)
```

usando **\?** no lo es:

```
import os
test = r'\\?\C:\Python7'
print os.listdir(test)
```

La única forma de solucionar el error es asegurandose que el path termina con un separador ya que el código python hace lo siguiente:

```
if (ch != SEP && ch != ALTSEP && ch != ':')
        namebuf[len++] = '/';
```

Lo que quiere decir que \?\C:\Python7 se convierte a \?\C:\Python7/ lo que es un problema ya que como ya hemos dicho la API de Windows no convertirá el / en . Este es un problema que se repite en varias partes del paquete os.

5.2 Lockeando archivos

Ciertos sistemas operativos (en concreto Windows) utilizan locks en los archivos, lo que puede ser un problema en ciertos casos. Realmente no hay mucho que como desarrolladores podamos hacer a parte de asegurarnos que siempre cerremos el file descriptor, o bien usando **with** o explícitamente llamando a **close**. Uno de los problemas con los locks es que Windows no provee ninguna API pública para averiguar si un archivo ya esta abierto por otro proceso. Digo publica porque el siguiente script utiliza una API no documentada de NT para hacerlo (no recomiendo usar este código para producción, es más un ejemplo divertido):

```
import os
import struct

import winerror
import win32file
```

```
import win32con

from ctypes import *
from ctypes.wintypes import *
from Queue import Queue
from threading import Thread
from win32api import GetCurrentProcess, OpenProcess, DuplicateHandle
from win32api import error as ApiError
from win32con import (
    FILE_SHARE_READ,
    FILE_SHARE_WRITE,
    OPEN_EXISTING,
    FILE_FLAG_BACKUP_SEMANTICS,
    FILE_NOTIFY_CHANGE_FILE_NAME,
    FILE_NOTIFY_CHANGE_DIR_NAME,
    FILE_NOTIFY_CHANGE_ATTRIBUTES,
    FILE_NOTIFY_CHANGE_SIZE,
    FILE_NOTIFY_CHANGE_LAST_WRITE,
    FILE_NOTIFY_CHANGE_SECURITY,
    DUPLICATE_SAME_ACCESS
)
from win32event import WaitForSingleObject, WAIT_TIMEOUT, WAIT_ABANDONED
from win32event import error as EventError
from win32file import CreateFile, ReadDirectoryChangesW, CloseHandle
from win32file import error as FileError

# from ubuntuone.platform.windows.os_helper import LONG_PATH_PREFIX, abspath

LONG_PATH_PREFIX = '\\\\?\\'
# constant found in the msdn documentation:
# http://msdn.microsoft.com/en-us/library/ff538834(v=vs.85).aspx
FILE_LIST_DIRECTORY = 0x0001
FILE_NOTIFY_CHANGE_LAST_ACCESS = 0x00000020
FILE_NOTIFY_CHANGE_CREATION = 0x00000040

# XXX: the following code is some kind of hack that allows to get the opened
# files in a system. The techinique uses an no documented API from windows nt
# that is internal to MS and might change in the future braking our code :(
UCHAR = c_ubyte
```

```
PVOID = c_void_p
```

```
ntdll = windll.ntdll

SystemHandleInformation = 16
STATUS_INFO_LENGTH_MISMATCH = 0xC0000004
STATUS_BUFFER_OVERFLOW = 0x80000005L
STATUS_INVALID_HANDLE = 0xC0000008L
STATUS_BUFFER_TOO_SMALL = 0xC0000023L
STATUS_SUCCESS = 0

CURRENT_PROCESS = GetCurrentProcess ()
DEVICE_DRIVES = {}
for d in "abcdefghijklmnopqrstuvwxyz":
    try:
        DEVICE_DRIVES[win32file.QueryDosDevice (d + ":").strip ("\x00").lower ()] = d + ":"
    except FileError, (errno, errctx, errmsg):
        if errno == 2:
            pass
        else:
            raise

class x_file_handles(Exception):
    pass

def signed_to_unsigned(signed):
    unsigned, = struct.unpack ("L", struct.pack ("l", signed))
    return unsigned

class SYSTEM_HANDLE_TABLE_ENTRY_INFO(Structure):
    """Represent the SYSTEM_HANDLE_TABLE_ENTRY_INFO on ntdll."""
    _fields_ = [
        ("UniqueProcessId", USHORT),
        ("CreatorBackTraceIndex", USHORT),
        ("ObjectTypeIndex", UCHAR),
        ("HandleAttributes", UCHAR),
        ("HandleValue", USHORT),
        ("Object", PVOID),
        ("GrantedAccess", ULONG),
    ]
```

```
class SYSTEM_HANDLE_INFORMATION(Structure):
    """Represent the SYSTEM_HANDLE_INFORMATION on ntdll."""
    _fields_ = [
        ("NumberOfHandles", ULONG),
        ("Handles", SYSTEM_HANDLE_TABLE_ENTRY_INFO * 1),
    ]
```

```
class LSA_UNICODE_STRING(Structure):
    """Represent the LSA_UNICODE_STRING on ntdll."""
    _fields_ = [
        ("Length", USHORT),
        ("MaximumLength", USHORT),
        ("Buffer", LPWSTR),
    ]

class PUBLIC_OBJECT_TYPE_INFORMATION(Structure):
    """Represent the PUBLIC_OBJECT_TYPE_INFORMATION on ntdll."""
    _fields_ = [
        ("Name", LSA_UNICODE_STRING),
        ("Reserved", ULONG * 22),
    ]

class OBJECT_NAME_INFORMATION (Structure):
    """Represent the OBJECT_NAME_INFORMATION on ntdll."""
    _fields_ = [
        ("Name", LSA_UNICODE_STRING),
    ]

class IO_STATUS_BLOCK_UNION (Union):
    """Represent the IO_STATUS_BLOCK_UNION on ntdll."""
    _fields_ = [
        ("Status", LONG),
        ("Pointer", PVOID),
    ]

class IO_STATUS_BLOCK (Structure):
    """Represent the IO_STATUS_BLOCK on ntdll."""
    _anonymous_ = ("u",)
    _fields_ = [
        ("u", IO_STATUS_BLOCK_UNION),
        ("Information", POINTER (ULONG)),
    ]

class FILE_NAME_INFORMATION (Structure):
    """Represent the on FILE_NAME_INFORMATION ntdll."""
    filename_size = 4096
```

```
    _fields_ = [
        ("FilenameLength", ULONG),
        ("FileName", WCHAR * filename_size),
    ]

def get_handles():
    """Return all the processes handles in the system atm."""
    system_handle_information = SYSTEM_HANDLE_INFORMATION()
    size = DWORD (sizeof (system_handle_information))
    while True:
        result = ntdll.NtQuerySystemInformation(
            SystemHandleInformation,
            byref(system_handle_information),
            size,
            byref(size)
        )
        result = signed_to_unsigned(result)
        if result == STATUS_SUCCESS:
            break
        elif result == STATUS_INFO_LENGTH_MISMATCH:
            size = DWORD(size.value * 4)
            resize(system_handle_information, size.value)
        else:
            raise x_file_handles("NtQuerySystemInformation", hex(result))

    pHandles = cast(
        system_handle_information.Handles,
        POINTER(SYSTEM_HANDLE_TABLE_ENTRY_INFO * \
                system_handle_information.NumberOfHandles)
    )
    for handle in pHandles.contents:
        yield handle.UniqueProcessId, handle.HandleValue

def get_process_handle (pid, handle):
    """Get a handle for the process with the given pid."""
    try:
        hProcess = OpenProcess(win32con.PROCESS_DUP_HANDLE, 0, pid)
        return DuplicateHandle(hProcess, handle, CURRENT_PROCESS,
            0, 0, DUPLICATE_SAME_ACCESS)
```

```
    except ApiError,(errno, errctx, errmsg):
        if errno in (
              winerror.ERROR_ACCESS_DENIED,
              winerror.ERROR_INVALID_PARAMETER,
              winerror.ERROR_INVALID_HANDLE,
              winerror.ERROR_NOT_SUPPORTED
        ):
            return None
        else:
            raise


def get_type_info (handle):
    """Get the handle type information."""
    public_object_type_information = PUBLIC_OBJECT_TYPE_INFORMATION()
    size = DWORD(sizeof(public_object_type_information))
    while True:
        result = signed_to_unsigned(
            ntdll.NtQueryObject(
                handle, 2, byref(public_object_type_information), size, None))
        if result == STATUS_SUCCESS:
            return public_object_type_information.Name.Buffer
        elif result == STATUS_INFO_LENGTH_MISMATCH:
            size = DWORD(size.value * 4)
            resize(public_object_type_information, size.value)
        elif result == STATUS_INVALID_HANDLE:
            return None
        else:
            raise x_file_handles("NtQueryObject.2", hex (result))


def get_name_info (handle):
    """Get the handle name information."""
    object_name_information = OBJECT_NAME_INFORMATION()
    size = DWORD(sizeof(object_name_information))
    while True:
        result = signed_to_unsigned(
            ntdll.NtQueryObject(handle, 1, byref (object_name_information),
            size, None))
```

```
        if result == STATUS_SUCCESS:
            return object_name_information.Name.Buffer
```

```
        elif result in (STATUS_BUFFER_OVERFLOW, STATUS_BUFFER_TOO_SMALL,
                        STATUS_INFO_LENGTH_MISMATCH):
            size = DWORD(size.value * 4)
            resize (object_name_information, size.value)
        else:
            return None


def filepath_from_devicepath (devicepath):
    """Return a file path from a device path."""
    if devicepath is None:
        return None
    devicepath = devicepath.lower()
    for device, drive in DEVICE_DRIVES.items():
        if devicepath.startswith(device):
            return drive + devicepath[len(device):]
    else:
        return devicepath

def get_real_path(path):
    """Return the real path avoiding issues with the Library a in Windows 7"""
    assert os.path.isdir(path)
    handle = CreateFile(
        path,
        FILE_LIST_DIRECTORY,
        FILE_SHARE_READ | FILE_SHARE_WRITE,
        None,
        OPEN_EXISTING,
        FILE_FLAG_BACKUP_SEMANTICS,
        None
    )
    name = get_name_info(int(handle))
    CloseHandle(handle)
    return filepath_from_devicepath(name)

def get_open_file_handles():
    """Return all the open file handles."""
    print 'get_open_file_handles'
    result = set()
```

```
    this_pid = os.getpid()
    for pid, handle in get_handles():
```

```
        if pid == this_pid:
            continue
        duplicate = get_process_handle(pid, handle)
        if duplicate is None:
            continue
        else:
            # get the type info and name info of the handle
            type = get_type_info(handle)
            name = get_name_info(handle)
            # add the handle to the result only if it is a file
            if type and type == 'File':
                # the name info represents the path to the object,
                # we need to convert it to a file path and then
                # test that it does exist
                if name:
                    file_path = filepath_from_devicepath(name)
                    if os.path.exists(file_path):
                        result.add(file_path)
    return result

def get_open_file_handles_under_directory(directory):
    """get the open files under a directory."""
    result = set()
    all_handles = get_open_file_handles()
    # to avoid issues with Libraries on Windows 7 and later, we will
    # have to get the real path
    directory = get_real_path(os.path.abspath(directory))
    if not directory.endswith(os.path.sep):
        directory += os.path.sep
    for file in all_handles:
        if directory in file:
            result.add(file)
    return result
```

5.3 Unicode, unicode, unicode

Cuando se trabaja con el file system en varias plataformas debemos de tener gran cuidado con el unicode de nuestros paths, nombre de usuarios etc., ya que puede hacer que nuestra aplicación se rompa. Hay varios casos en los que tenemos que tener un gran cuidado.

expanduser

En python 2.7 hay un bug por que el que si el usuario utiliza caracteres unicode en Windows no funciona correctamente. Un ejemplo, si nuestro usuario es Japones y su nombre es "□□□□□□" lo siguiente se encontrara el el sistema:

- El shell de Windows (Explorer) mostrara correctamente "C:Users□□□□□□"
- cmd.exe mostrara "C:Users??????"
- Todas las variables de sistema serán equivocadas.

expanduser esta roto por que utiliza las variables del sistema para expandir el path. El motivo exacto se debe a que posixmodule.c en python 2.7 utiliza PyString_FromString*() cuando debería de utilizar _wenviron PyUnicode_FromWideChar() (cosa que se hace en python 3), una forma de solucionarlo es usar ctypes de la siguiente manera:

```
import ctypes
from ctypes import windll, wintypes

class GUID(ctypes.Structure):
    _fields_ = [
         ('Data1', wintypes.DWORD),
         ('Data2', wintypes.WORD),
         ('Data3', wintypes.WORD),
         ('Data4', wintypes.BYTE * 8)
    ]

    def __init__(self, l, w1, w2, b1, b2, b3, b4, b5, b6, b7, b8):
        """Create a new GUID."""
        self.Data1 = l
        self.Data2 = w1
        self.Data3 = w2
        self.Data4[:] = (b1, b2, b3, b4, b5, b6, b7, b8)
```

```
    def __repr__(self):
        b1, b2, b3, b4, b5, b6, b7, b8 = self.Data4
        return 'GUID(%x-%x-%x-%x%x%x%x%x%x%x%x)' % (
                   self.Data1, self.Data2, self.Data3, b1, b2, b3, b4, b5, b6, b7, b8)

# constants to be used according to the version on shell32
CSIDL_PROFILE = 40
FOLDERID_Profile = GUID(0x5E6C858F, 0x0E22, 0x4760, 0x9A, 0xFE, 0xEA, 0x33, 0x17, 0xB6, 0x71, 0x73)

def expand_user():
    # get the function that we can find from Vista up, not the one in XP
    get_folder_path = getattr(windll.shell32, 'SHGetKnownFolderPath', None)
    if get_folder_path is not None:
        # ok, we can use the new function which is recomended by the msdn
        ptr = ctypes.c_wchar_p()
        get_folder_path(ctypes.byref(FOLDERID_Profile), 0, 0, ctypes.byref(ptr))
        return ptr.value
    else:
        # use the deprecated one found in XP and on for compatibility reasons
       get_folder_path = getattr(windll.shell32, 'SHGetSpecialFolderPathW', None)
       buf = ctypes.create_unicode_buffer(300)
       get_folder_path(None, buf, CSIDL_PROFILE, False)
       return buf.value
```

mbcs

En mbcs es el *multibyte encoding* que utiliza Windows en su file system. Un *multibyte encodings* tiene un número variable de bytes para representar los caracteres por lo que se ha de tener especial cuidado si vamos a intercambiar información sombre paths de un sistem Windows a otro.

caracteres illegales

Un usuario naive de Linux no tiene el concepto de caracteres ilegales, por lo que es posible que si utiliza la misma aplicación en Windows y en Linux quiera utilizar paths ilegales en la otra plataforma. Realmente no hay forma arreglar esto, los caracteres ilegales son ilegales y punto. Lo que si que podemos hacer es ser es ligeramente astutos (realmente fue Chipaca el que propuso esta solución). La idea es sencilla, antes de escribir un archivo en Windows vamos a reemplazar los caracteres ilegales del path con caracteres unicode que sean muy parecidos a los originales, este es un ejemplo de dicho map:

```
BASE_CODE = u'\N{ZERO WIDTH SPACE}%s\N{ZERO WIDTH SPACE}'
WINDOWS_ILLEGAL_CHARS_MAP = {
    u'<': BASE_CODE % u'\N{SINGLE LEFT-POINTING ANGLE QUOTATION MARK}',
```

```
    u'>': BASE_CODE % u'\N{SINGLE RIGHT-POINTING ANGLE QUOTATION MARK}',
    u':': BASE_CODE % u'\N{RATIO}',
    u'"': BASE_CODE % u'\N{DOUBLE PRIME}',
    u'/': BASE_CODE % u'\N{FRACTION SLASH}',
    u'|': BASE_CODE % u'\N{DIVIDES}',
    u'?': BASE_CODE % u'\N{INTERROBANG}',
    u'*': BASE_CODE % u'\N{SEXTILE}',
}
# inverse map
LINUX_ILLEGAL_CHARS_MAP = {}
for key, value in WINDOWS_ILLEGAL_CHARS_MAP.iteritems():
    LINUX_ILLEGAL_CHARS_MAP[value] = key
```

Aunque los caracteres no son exactamente los mismos son los suficientemente parecidos, ademas el usuario no tiene la posibilidad de compararlos con los originales ya que son ilegales y no están presentes en el sistema ;-).

5.3 chmod

Por desgracia, aunque el paquete os nos proporciona chmod su funcionamiento no es el adecuado en todas las plataformas. De nuevo Windows nos propone un problema que resolver, ya que lo que hace chmod es simplemente tocar los flags de seguridad del archivo pero no los ACE. Mientras que en Mac OS X y Linux podemos confiar en chmod, en Windows tenemos que utilizar su API de C para modificar los ACE. Podemos añadir un ACE que da más poderes al usuario:

```
def add_ace(path, rights):
    """Remove rights from a path for the given groups."""
    if not os.path.exists(path):
        raise WindowsError('Path %s could not be found.' % path)

    if rights is not None:
        security_descriptor = GetFileSecurity(path, DACL_SECURITY_INFORMATION)
        dacl = security_descriptor.GetSecurityDescriptorDacl()
        # set the attributes of the group only if not null
        dacl.AddAccessAceEx(ACL_REVISION_DS,
                CONTAINER_INHERIT_ACE | OBJECT_INHERIT_ACE, rights,
                USER_SID)
        security_descriptor.SetSecurityDescriptorDacl(1, dacl, 0)
```

```
        SetFileSecurity(path, DACL_SECURITY_INFORMATION, security_descriptor)


def remove_ace(path):
    """Remove the deny ace for the given groups."""
    if not os.path.exists(path):
        raise WindowsError('Path %s could not be found.' % path)
    security_descriptor = GetFileSecurity(path, DACL_SECURITY_INFORMATION)
    dacl = security_descriptor.GetSecurityDescriptorDacl()
    # if we delete an ace in the acl the index is outdated and we have
    # to ensure that we do not screw it up. We keep the number of deleted
    # items to update accordingly the index.
    num_delete = 0
    for index in range(0, dacl.GetAceCount()):
        ace = dacl.GetAce(index - num_delete)
        # check if the ace is for the user and its type is 0, that means
        # is a deny ace and we added it, lets remove it
        if USER_SID == ace[2] and ace[0][0] == 0:
            dacl.DeleteAce(index - num_delete)
            num_delete += 1
    security_descriptor.SetSecurityDescriptorDacl(1, dacl, 0)
    SetFileSecurity(path, DACL_SECURITY_INFORMATION, security_descriptor)
```

o uno que se los quita:

```
def add_deny_ace(path, rights):
    """Remove rights from a path for the given groups."""
    if not os.path.exists(path):
        raise WindowsError('Path %s could not be found.' % path)

    if rights is not None:
        security_descriptor = GetFileSecurity(path, DACL_SECURITY_INFORMATION)
        dacl = security_descriptor.GetSecurityDescriptorDacl()
        # set the attributes of the group only if not null
        dacl.AddAccessDeniedAceEx(ACL_REVISION_DS,
                CONTAINER_INHERIT_ACE | OBJECT_INHERIT_ACE, rights,
                USER_SID)
        security_descriptor.SetSecurityDescriptorDacl(1, dacl, 0)
        SetFileSecurity(path, DACL_SECURITY_INFORMATION, security_descriptor)
```

```
def remove_deny_ace(path):
    """Remove the deny ace for the given groups."""
    if not os.path.exists(path):
        raise WindowsError('Path %s could not be found.' % path)
    security_descriptor = GetFileSecurity(path, DACL_SECURITY_INFORMATION)
    dacl = security_descriptor.GetSecurityDescriptorDacl()
    # if we delete an ace in the acl the index is outdated and we have
    # to ensure that we do not screw it up. We keep the number of deleted
    # items to update accordingly the index.
    num_delete = 0
    for index in range(0, dacl.GetAceCount()):
        ace = dacl.GetAce(index - num_delete)
        # check if the ace is for the user and its type is 1, that means
        # is a deny ace and we added it, lets remove it
        if USER_SID == ace[2] and ace[0][0] == 1:
            dacl.DeleteAce(index - num_delete)
            num_delete += 1
    security_descriptor.SetSecurityDescriptorDacl(1, dacl, 0)
    SetFileSecurity(path, DACL_SECURITY_INFORMATION, security_descriptor)
```

6. Los procesos también tienen vida social

A la hora de diseñar nuestra aplicación tenemos que tener en cuenta que el Inter Process Communication de cada plataforma es diferente y tiene diferentes restricciones. Mientras que programadores Windows evitan tener que hacer IPC, ya que tienen que diseñar el protocolo y no está muy claro cuál es el mejor camino, aquellos que trabajamos en Linux tenemos dbus el cual realiza gran parte del trabajo para nosotros. Independientemente de la implementación que se tome, siempre podemos intentar reducir el código al máximo. La idea es sencilla, si tenemos que hacer IPC diseñemosló como una API que se va a utilizar desde nuestro paquete. Una vez que estemos contentos con la API la expondremos de la forma adecuada según la plataforma.

Lo que permite este diseño es compartir la mayor cantidad de lógica entre las plataformas (ya que la lógica de nuestra aplicación es lo más complejo) y testearla sin necesidad de hacer tests de integración (los cuales haremos según en que plataforma nos encontremos).

8. Conclusión

A pesar de que Python en si es multiplataforma, a la hora de trabajar con más de un sistema tenemos que tener cuidado con aquellas áreas en las que estemos en contacto con el SO. El trabajo es laborioso pero es posible conseguir una aplicación que se comporte de la misma forma en el 90% de los casos, y el resto de los casos.. bueno ya se vera que hacemos :-)

Finanzas cuantitativas con Python

Autor: Damián Avila

Bio: Lic. Bioquímica, Doctorando en Ciencias Biológicas, Docente Universitario, “Analista Cuantitativo” (in progress...)

Web: http://damianavila.blogspot.com

Email: damianavila@gmail.com

Twitter: @damian_avila

En los últimos 10 años se han desarrollado diversos proyectos que hacen de Python un **“instrumento de elección”** para la investigación en finanzas cuantitativas. En esta pequeña reseña, nos proponemos no sólo hacer una breve descripción de las herramientas disponibles en Python para el análisis y el modelado de las series de tiempo, sino también presentar la integración de estas herramientas en un **“ambiente pythonico”** donde podemos realizar nuestras investigaciones de una manera simple y eficiente.

¿Por qué usar Python para resolver problemas en finanzas cuantitativas?

Existen múltiples plataformas sobre las cuales pueden desarrollarse modelos estadístico-matemáticos y aplicarlos a los mercados financieros. Entre ellos podemos citar plataformas comerciales (MATLAB, Eviews, Stata, SPSS, SAS), plataformas gratuitas de código cerrado (EasyReg) y plataformas gratuitas de código abierto (R y Gretl). Frente a tantas opciones: **¿Por qué elegir Python?**

Python posee diversas características que lo transforman en una excelente opción para el modelado cuantitativo de los mercados financieros. Es un lenguaje interpretado (puede utilizarse interactivamente), es gratuito y de código abierto, posee múltiples “baterías adicionales” que lo potencian enormemente y, finalmente, Python es fácil a aprender y usar (de yapa... ¡Es divertido!).

¿Cuales son las “baterías adicionales” necesarias para las finanzas cuantitativas?

- **Numpy** (http://numpy.scipy.org): Extiende Python con un nuevo objeto, a saber, los arreglos multidimensionales (con métodos avanzados de corte y reorganización). Además contiene 3 librerías con rutinas numéricas (álgebra linear, Fourier y números aleatorios) y está escrito mayoritariamente en C (por lo que puede extenderse con código escrito en C y Fortran).
- **Scipy** (http://www.scipy.org): Extiende Python con herramientas computacionales para científicos. Está desarrollado sobre Numpy y provee módulos para estadística, optimización, integración numérica, álgebra lineal, transformadas de Fourier, procesamiento de señales, procesamiento de imágenes, resolución de ODE y otras funciones especiales específicas.
- **Matplotlib** (http://matplotlib.sourceforge.net): Extiende Python con gráficos 2D de gran calidad. Se pueden generar diferentes tipos de gráficos: puntos, lineas, barras, histogramas, espectros, etc. Además poseemos un control total sobre el estilo, las fuentes y los ejes, a través de un conjunto de funciones tipo Matlab (o, a través de una interfase orientada a objetos).
- **IPython** (http://ipython.org): IPython facilita el uso de Python de manera interactiva a través de consolas que soportan la visualización de datos y el completado por *<TAB>*. Además es una plataforma de procesamiento en paralelo.
- **Statsmodels** (http://statsmodels.sourceforge.net): Extiende Python con un conjunto de clases y funciones para la estimación de múltiples modelos econométricos (provee una detallada lista de estadísticos para cada estimador). Permite la exploración descriptiva de los datos y la inferencia a través de la realización de múltiples test.

  Algunas de las herramientas emblemáticas disponibles en Statsmodels:

  - Regresión: OLS, GLS (incluyendo WLS y AR-LS).
  - GLM: modelos lineales generalizados.
  - RLM: modelos lineales robustos.
  - Discretemod: Regresión con variables dependientes discretas, Logit, Probit, MNLogit y Poisson, basados en estimadores de máxima verosimilitud.

- Análisis de series de tiempos univariable: AR, ARIMA.
- Estadísticas descriptiva y modelos de procesos.
- VAR: modelos de vectores auto-regresivos.
- Datasets: ejemplos y testeo.

¿Cómo podemos integrar todas estas herramientas disponibles?

Una de las posibles respuestas: Spyder (http://packages.python.org/spyder)

Spyder (Scientific PYthon Development EnviRonment) es, no sólo un "environment" de desarrollo con herramientas avanzadas de edición, testeo interactivo y debugging, sino también, una plataforma para cálculo científico (no sólo restringido a las finanzas cuantitativas) gracias a la integración de Numpy, Scipy, Matplotlib e IPython en una "lugar común" al alcance de todos.

Un pequeño ejemplo...

Ahora que conocemos algunas de las herramientas disponibles, un poco de acción...

¿Qué es un cupón atado al PBI?

Es un instrumento financiero que cotiza en la bolsa y que paga (a año vencido) un "excedente" que depende (esencialmente) del PBI real, la inflación y el tipo de cambio.

¿Es el TVPP (uno de los cupones atados al PBI) una alternativa de inversión interesante a mediano-largo plazo?

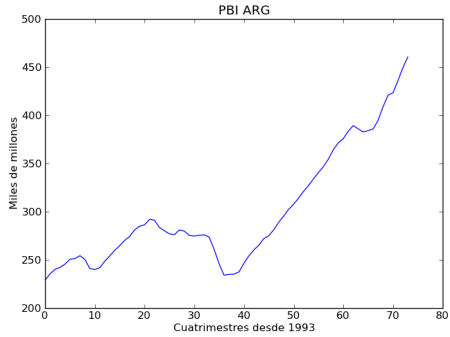
Para responder esa pregunta trataremos de modelar la evaluación temporal del PBI (el principal determinante del pago) a través de un proceso auto-regresivo lineal (univariado).

NOTA_1: Se darán por conocidos algunos conceptos básicos del análisis de series de tiempo con el objetivo de ejemplificar un tipo de análisis que "normalmente" se realiza en finanzas cuantitativas.

Para ello, "dibujamos" la serie (que obtuvimos previamente online) usando funciones de matplotlib:

```
import matplotlib.pyplot as plt
```

```
TVPP = …

plt.figure()
plt.title(PBI ARG)
plt.xlabel("Cuatrimestres desde 1993")
plt.ylabel("Miles de millones")
plt.plot(TVPP.close)

plt.show()
```

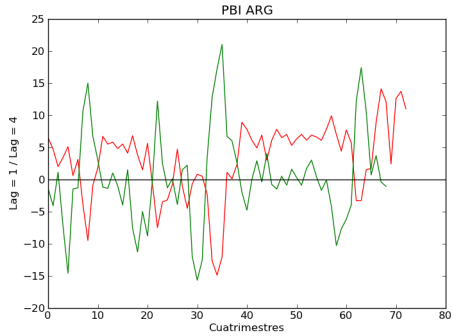


Asimismo, diferenciamos en orden 1 (linea roja) y orden 4 (linea verde) para obtener retornos y desestacionalizar, respectivamente.

```
ret_close = [] # retornos (lag_1)

for i in range(1, n-1):
    ret_close.append(TVPP.close[i] - TVPP.close[i-1])
```

```
des_close = [] # retornos desestacionalizados (lag_4)

for i in range(4, n-2):
    des_close.append(ret_close[i] - ret_close[i-4])
```

```
ar_des_close = np.array(des_close) # convertir lista en np.array
```

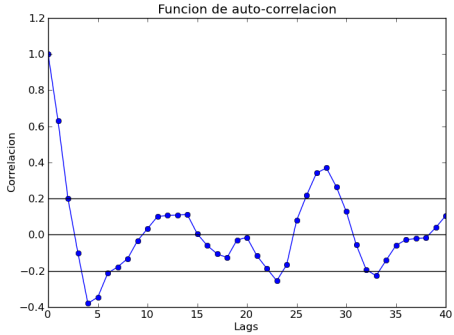


Luego, verificamos la obtención de una serie de tiempo débilmente **ESTACIONARIA** (ADF test).

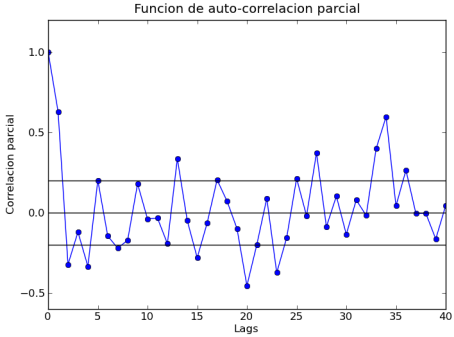
```
adftest = sm.tsa.adfuller(ar_des_close, None, "c")

(-4.5919718832856455, 0.00013364295285426029, 2, 66,
{'5 %': -2.9064436883991434, '1 %': -3.5335601309235605,
'10 %': -2.590723948576676}, 359.65601420392511)
```

Calculamos ACF y PACF (funciones de autocorrelación y autocorrelación parcial) para determinar algunas características esenciales del modelo (y que tienen que ver con la información "subyacente" en los "valores previos - lags").

```
auto_corrtest = sm.tsa.acf(ar_des_close)
```



```
pauto_corrtest = sm.tsa.pacf_ols(ar_des_close)
```



Finalmente, estimamos los parámetros del modelo por “fiteo” de la serie de tiempo.

```
ARmodel = sm.tsa.AR(ar_des_close)
ARmodel_fit = ARmodel.fit(maxlag = 4)
```

Con el modelo “fiteado” podemos pasar a la siguiente fase: el “forecasting” (es decir, tratar de predecir cómo evolucionará el PBI en los próximos trimestres).

```
ARmodel_fore = ARmodel.predict(n = 10, start = -1)

[-4.57970687 0.02294563 -0.52401676 -0.05539513
0.90010725 1.02408095 1.06120815 1.04202971 0.71406367
0.43024225]
```

Así llegamos a “forecastear” el PBI 2011 y PBI 2012 en **~10,5%** y **~9,2 %**.

NOTA_2: Para valuar el cupón existen otras consideraciones teóricas adicionales que no se tratarán en este artículo.

Teniendo en cuenta que el valor del TVPP (x100) en el mercado (al 22-09-2011, el día previo a la PyConAr 2011) fue de $14,78, considerando el PBI registrado en el 2010 y los PBI que hemos proyectado para 2011 - 2012, tendríamos la siguiente progresión de pagos:

- **15/12/2011:** $5,97 (asegurado)

- **15/12/2012:** $9,61 (muy probable)
- **15/12/2013:** $14,03 (probable)

Por lo tanto, con los próximos dos pagos pagos lograríamos cubrir la cotización de los cupones en el mercado al 22-09-2011 ($5,97 + $9,61), y todos los pagos posteriores (si los hubiere) serían "libres de riesgo". Podemos concluir entonces que en el mediano-largo plazo, la compra de cupones atados al PBI parece un opción atractiva.

NOTA_3: Este ejemplo NO es una recomendación de compra de activos financieros, sólo se trata de un ejercicio para demostrar las funcionalidades econométricas de las librerías que hemos mencionado previamente.

Conclusiones finales

¿Tenemos un "environment" completo en Python para realizar investigación en finanzas cuantitativas?

En la actualidad, tenemos un "environment" amplio, potente y, además, con mucho potencial, a través, no sólo de mejoras en la "baterías " que hemos reseñado, sino también, a partir de la posibilidad de integración de nuevas "baterías" que agregarán nuevas funcionalidades específicas para el área.

Más datos sobre el autor

Damian Avila es analista en finanzas cuantitativas, pero también Licenciado en Bioquímica (doctorando en ciencias biológicas) de la UNLP.

Utiliza Python (con baterías adicionales específicas) para el análisis de series de tiempo financieras y para la construcción de modelos matemáticos en biología.

Es docente universitario en la Cátedra de Inmunología en la FCE - UNLP y dicta cursos de Bioestadística en el Colegio de Bioquímicos - Colegio Zonal XII - Buenos Aires.

Es miembro de Python Argentina (PyAr - http://python.org.ar/pyar/) y del Quantitative Finance Club (QFClub – http://qfclub.wordpress.com/).

xkcd

La sabiduría de los antiguos

Este cómic proviene de xkcd, un comic web de romance, sarcasmo, matemática y lenguaje (http://xkcd.com)