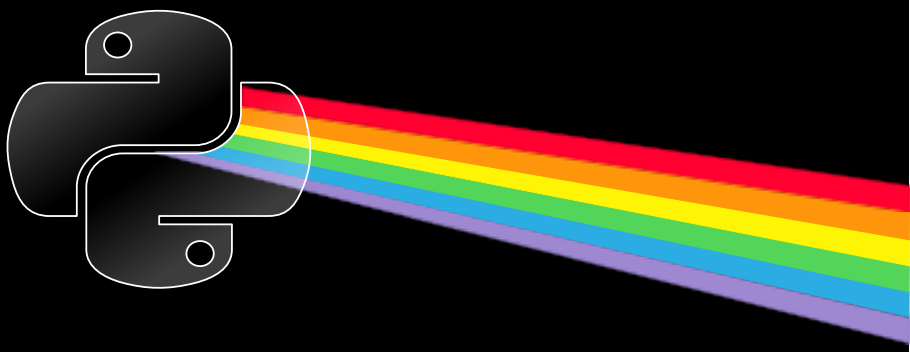


PET #6 - Nov 2012 - ISSN: 1853-2071

Licencia

En Este Número

Staff

Editores: Juan B Cabral **Corrector:** Walter Alini **Sitio:** http://revista.python.org.ar

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. Portada: Catriel Müller

Editor responsable: Roberto Alsina, Don Bosco 146 Dto 2, San Isidro, Argentina. **ISSN:** 1853-2071

Con perdón de Cervantes

En algún momento de diciembre cuya exactitud no quiero acordarme, reposaban dos hidalgos (uno en Córdoba y otro en Castelar) con amplia calma y muy buen pasar.

A uno de ellos, por leer libros de fantasía, ocurriósele la empresa de organizar la 3era kermés más grande del lenguaje Python del 2012, y el otro en su ignorancia lo acompañó.

Molinos de por medio, 11 meses después de su partida estos dos valientes confían en que la épica no les dará la espalda y contribuyen con estos versos a lograr que vuestras mercedes se vean nutridos por tan magno esfuerzo.

Juan B Cabral - Editor Pet Nro.6 - Co-coord. Gral. PyCon 2012

El Ignorante, el Quijote y el Valentón

Empaquetando aplicaciones Django

Autor: Daniel F. Moisset

Bio: Desarrollador de Python veterano, emprendedor y domador de serpientes en Machinalis, docente de CS y ñoño orgulloso.

Web: http://machinalis.com

Email: dmoisset@machinalis.com

Twitter: @dmoisset

Django permite hacer aplicaciones reusables fácilmente, y es uno de los frameworks con más aplicaciones de terceros disponibles. Pero muchos autores de aplicaciones no las comparten o lo hacen de una forma impráctica para los usuarios simplemente porque publicarlas en la forma convencional requiere seguir una serie de pasos poco documentada. Este artículo permite pasar de ser un autor de aplicación a publicarla en algunos minutos.

Escribo este artículo no como un experto en empaquetado, sino como un novato que documenta su experiencia. En especial, considerando que la documentación es algo confusa y no se enfoca en aspectos puntuales relacionados con Django.

En este articulo estoy suponiendo que escribiste una aplicación muy piola para Django y que te gustaría que tus usuarios pudieran bajarla e instalarla desde PyPI (por ejemplo corriendo *pip install django-fancy-app*) y que pueden instalar la versión más reciente del repositorio de control de código (por ejemplo `pip install -e https://github.com/lectordepet/django-fancy-app.git`). También voy a suponer que tenés algo de experiencia básica programando Python.

Algunas de las explicaciones incluidas son completamente genéricas al empaquetado de cualquier cosa en Python, pero algunos detalles y supuestos son específicos para aplicaciones Django.

¡Opciones! ¡Opciones!

Lo primero que vas a darte cuenta si googleás un poco sobre cómo empaquetar en Python es que hay muchas opciones:

- *distutils*
- *distutils2*
- *distribute*
- *setuptools*

Si todavía no te parecen demasiadas, capaz también hayas escuchado de un módulo llamado “packaging” (que es en realidad lo mismo que distutils2, que será renombrado al incorporarse a la biblioteca estándar de Python 3). Por lo que parece, cada uno de ellos fue escrito porque «los otros eran muy limitados» o algo por el estilo.

En este artículo voy a estar usando distutils. Lo que me gusta de distutils es que viene incluido en la biblioteca estándar de Python 2.x (que si estás desarrollando para Django, es la versión que estás usando), y para aplicaciones Django normalmente es lo suficientemente bueno. Aparentemente el panorama es que todo está convergiendo hacia distutils2, que supuestamente va a ser compatible hacia atrás una vez que se incorpore (bajo el nombre “packaging”) en Python 3. Para más información, te recomiendo leer <http://guide.python-distribute.org/introduction.html#current-state-of-packaging>

Primeros pasos

Lo primero que necesitás hacer es decidir dos nombres. El primero de ellos es el *nombre en PyPI*, es decir el nombre que tus usuarios van a escribir al hacer `pip install …`. El segundo es el *nombre en Python*, que tiene que ser un identificador valido en Python (alfanumérico y guiones bajos, sin espacios, etc.), que es lo que tus usuarios van escribir dentro del código al poner `import …`, y/o lo que agregarán en la opción `INSTALLED_APPS` de su proyecto.

No hay una convención única para estos nombres. La que a mí me gusta son palabras separadas por guión común, con un prefijo `django-` para el nombre en PyPI; y palabras separadas con guión bajo para nombres en Python, sin el prefijo `django`. En el ejemplo de este artículo vamos a seguir esta convención, con lo cual el nombre en PyPI será `django-fancy-app` y el nombre en Python va a ser `fancy_app`.

La organización de directorios ideal para empaquetar es tener tu aplicación en un directorio llamado con el nombre en Python. Este directorio debería quedar un nivel adentro de la raíz de tu árbol de control de revisiones. De este modo el directorio raíz queda reservado para archivos de control que son usados por las herramientas de empaquetado, archivos `README`, directorios de documentación, etc.

Necesitás crear dos archivos en la raíz para que funcione el sistema de empaquetado: uno llamado `setup.py` y otro llamado `MANIFEST.in`. El segundo no es un requisito el 100% del tiempo, pero casi seguro vas a necesitarlo. Además, el proceso de crear paquetes va a crear más archivos y carpetas llamados:

- `build/`.
- `dist/`.
- `django_fancy_app.egg-info/` (esto usa el nombre PyPI con guiones bajos en vez de altos, para confundir un poco más).
- `MANIFEST`.

así que asegurate que estos nombres estén disponibles (es decir, no crees archivos o directorios con estos nombres, y no los pongas en tu repositorio de control de revisiones -lo más probable es que quieras decirle que ignore estos archivos-).

Escribiendo los archivos de control

La mayoría de los archivos `setup.py` se ven así:

```
from distutils.core import setup

setup(
    ... # muchos argumentos nombrados que usa setup()
)
```

Este archivo es en efecto un script Python, y sigue la sintaxis estándar de cualquier programa en Python. Esta es la lista de los argumentos nombrados que normalmente te van a hacer falta:

`name`:

El nombre en PyPI de tu aplicación

`version`:

Una cadena de la forma *'1.2'* o *'0.3.1'*. La forma de esta cadena no es del todo libre, ya que se usa para comparar versiones cuando se instala desde archivos de requerimientos, por ejemplo si el archivo `requirementes.txt` del usuario indica `django-fancy-app>=1.3`

`description`:

Una descripción de un renglón, que se usa en los listados del sitio de PyPI

`long_description`:

Una descripción más detallada. Esto se va a ver en la página de tu aplicación dentro de PyPI. Podés usar texto formateado en ReST. En una sección más adelante, muestro algunos trucos útiles para llenar este campo.

`author`:

Tu nombre

`author_email`:

Tu dirección de correo electrónico

`url`:

Una URL que va a mostrarse en las descripciones del proyecto. Algunas personas apuntan aquí a una página web del proyecto, otros un enlace a un repositorio público como los de github o bitbucket.

`classifiers`:

Esta es una lista de etiquetas para herramientas que hacen índices de paquetes. Para las aplicaciones Django, casi seguro que querés ponerle al menos las siguientes:

- `Environment :: Web Environment`,
- `Intended Audience :: Developers`,
- `Framework :: Django`,
- `Programming Language :: Python`,
- tal vez `Topic :: Internet :: WWW/HTTP :: Dynamic Content`,
- Uno de los clasificadores de `Development Status` y uno de `License`

Hay un listado completo en <http://pypi.python.org/pypi?%3Aaction=list_classifiers>

`packages`:

Esta es una lista de directorios con archivos Python que se van a instalar. En el caso más simple, esto va a ser *['fancy_app']*. Pero si tenés más directorios o una jerarquía anidada, necesitás agregar todos los nodos. Por ejemplo, si *fancy_app* tiene un subdirectorio *views* con muchos archivos Python adentro, vas a tener que usar *['fancy_app', 'fancy_app.views']*

`zip_safe`:

Si tu aplicación tiene plantillas o datos estáticos, asignale *False*. Esta opción no tiene nada que ver con distribuir o no tu paquete en un formato comprimido (lo que sucede siempre), sino que está para permitir a la herramienta de empaquetado que instale la aplicación como un archivo comprimido en ZIP luego de ser descargado en el sistema del usuario. Python sabe importar módulos desde un archivo ZIP, con lo cual esto va a funcionar bien si tu aplicación es sólo código, pero Django no va a poder cargar las plantillas y archivos estáticos si no están directamente disponibles y descomprimidos en el disco. Configurar *zip_safe=False* garantiza que tu aplicación siempre sea descomprimida en el sistema del usuario al ser instalada, evitando este problema.

`include_package_data`:

Si tu aplicación tiene plantillas o datos estáticos, esta opción debe valer *True*. Esto indica que los archivos que no son Python sean instalados en el mismo lugar donde se instala la aplicación, que es donde Django sabe encontrarlos. Para que esto sirva de algo además es necesario dar un valor para la siguiente opción":

`package_data`:

Asignale `{ 'fancy_app': […] }`. La lista debería contener rutas relativas a `fancy_app`, y se pueden usar comodines de shell. Por ejemplo podrías tener `['templates/*.html', 'static/*.css', 'static/*.js']`

Por supuesto hay muchas otras opciones que podés consultar en la documentación de distutils en <http://docs.python.org/distutils/apiref.html#module-distutils.core>

El otro archivo que tenés que escribir es el *MANIFEST.in*, que se escribe en un minilenguaje hecho a medida. El propósito de este archivo es describir cuáles archivos incluir en el paquete. El sistema de empaquetado es lo suficientemente astuto como para descubrir los archivos indicados por setup.py: archivos Python indicados en *packages* y archivos descriptos en *package_data*, pero a veces puede que quieras incluir más archivos (por ejemplo un *README* un *LICENSE*, y documentación). Si ese es el caso, podés escribir algo como:

```
include README
include LICENSE
recursive-include doc *.rst *.png
```

Hay algunas otras directivas posibles, pero el ejemplo de arriba debería cubrirte en el 90% de los casos.

Agregando una descripción

El argumento *long_description* de setup se puede llenar con una cadena multilínea de Python común y corriente, formateada con ReST. Pero mantener documentación dentro del código fuente de un archivo Python no parece la mejor idea.

Un truco que he visto que se usa es tener un archivo README, y luego cargarlo al correr el script de setup. Es decir, añadir lo siguiente a tu *setup.py*:

```
...
with open('README') as readme:
    __doc__ = readme.read()

setup(
    ...
    long_description=__doc__,
    ...
)
```

Eso te permite mantener la descripción larga (que es, por cierto, el contenido de la página de PyPI para tu app) en un archivo de texto común, y mantener tu script de setup sincronizado automáticamente sin tener que repetir. También tiene el efecto colateral agradable de hacer que este texto sea el “docstring” (cadena de documentación) de tu `setup.py`.

Verificando que funcione

La mejor forma de comprobar que esto funcione correctamente es tener un pequeño proyecto de prueba, e instalar tu aplicación en un entorno “limpio”. La mejor forma de obtener un entorno limpio es usar `virtualenv`. Si nunca lo hiciste, deberías.

Con un virtualenv limpio activado, entrá al directorio donde tenés el setup.py y corré:

```
setup.py install
```

Si no funciona, notá que no hay una forma estándar de re-limpiar tu entorno. Si sólo has usado las herramientas que se describen en este artículo, borrar el directorio en *tuvirtualenv/lib/python2.x/site-packages/fancy_app/* debería alcanzar. Si querés estar 100% seguro, simplemente recreá el entorno virtual desde cero.

No uses pip para probar tu paquete. Cuando lo usás con un paquete en tu sistema de archivos, pip funciona un poco distinto que cuando descargás un paquete (en particular, en cuáles archivos copia), lo que puede resultar en que algunos problemas de tu paquete pasen desapercibidos. Si tu paquete funciona con *setup.py install*, sabés que después va a funcionar con pip, pero lo recíproco no sucede.

Subiendo todo a PyPI

En este momento, si tenés un repositorio público, tus usuarios ya deberían poder instalar tu paquete de control de revisiones usando *pip -e dirección-repo/django-fancy-app*.

Si ya te quedaste contento con el empaquetado, probablemente quieras compartir tu fabulosa aplicación vía PyPI, para facilitar a tus usuarios encontrarla, y además para poder proveer versiones específicas (o sea, las estables). Para poder hacer esto necesitás ser un usuario de PyPI; visitando http://pypi.python.org/ y haciendo click en “register” podés seguir las instrucciones para darte de alta.

Una vez que tenés un usuario, hace falta registrar el paquete. Aún cuando se puede hacer vía la interfaz por la web, la forma más fácil es correr *setup.py register*. Este comando te hace algunas preguntas de forma interactiva (incluyendo tu usuario y contraseña en PyPI) y va a dar de alta el paquete. El proceso de registro sólo hace falta hacerlo una vez (aún si luego subís muchas versiones).

Luego de eso, y cada vez que liberes una nueva versión, podés correr *setup.py sdist upload*. Cada vez que subís esto deberías tener un número de versión que aún no haya sido subido, sino PyPI va a protestar. Podrías borrar versiones desde el sitio web de PyPI, pero si ya hay usuarios que hayan bajado esa versión y la cambiás o borrás, se van a enojar mucho con vos. No lo hagas.

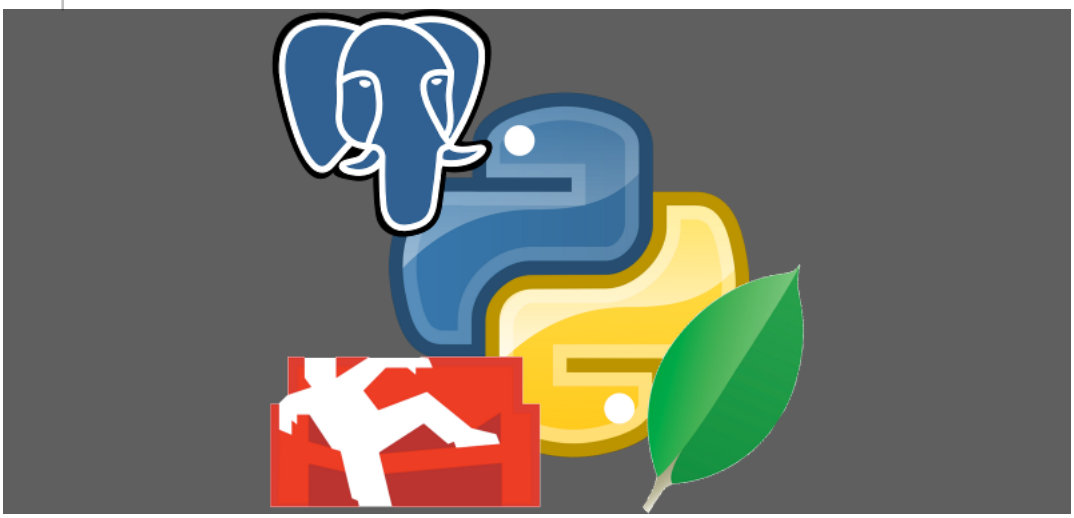
Con esto, creo que he delineado la mayoría de los puntos importantes. La documentación disponible para todo este proceso no es mala, pero hay mucha, y gran parte de ella se dedica a cosas que una app Django no necesita (por ejemplo distribuir ejecutables, o compilar extensiones en C), y es fácil pasar por alto detalles importantes y acabar con un paquete roto.

Pgpymongo y Pgpycouch: Extensiones para PostgreSQL en plpython para acceder a bases de datos NoSQL documentales

Autor: Anthony R. Sotolongo León

Bio: Ingeniero Informático desde el 2006, graduado en la Universidad de Cienfuegos, Miembro de la comunidad cubana de PostgreSQL, ha impartido cursos sobre el uso y explotación del gestor y ha desarrollado herramietas para el uso y enseñanza de PostgreSQL.

Email: asotolongo@uci.cu



MongoDB y CouchDB son las bases de datos NoSQL orientadas a documentos de código abierto más populares [10]. Su información se almacena en forma de *documentos* utilizando la Notación de Objetos JavaScript (JSON, por sus siglas en inglés). Cada documento simula una fila de una tabla de los sistemas relacionales, pero la diferencia es que son libres de esquema. Algunas empresas y entidades las están usando, como el caso de la Organización Europea de Investigaciones Nucleares con couchdb [11]. Springer y MTV usan mongodb [12].

Algunas características generales de cada gestor NoSQL

MongoDB:

- Lenguaje: C++
- Objetos: Colecciones y documentos (json)
- Replicación: Maestro - Esclavo
- Archivos: gridFS
- Acceso: TCP/IP

CouchDB

- Lenguaje: Erlang
- Objetos: Vistas y documentos (json)
- Replicación: Maestro - Maestro
- Archivos: Attachments
- Acceso: HTTP

Extensiones para PostgreSQL

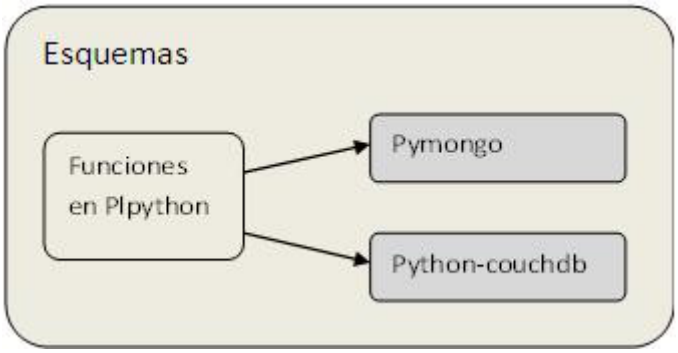
Las extensiones en PostgreSQL permiten agregar al gestor nuevas funcionalidades. Suelen incluir múltiples objetos de SQL; por ejemplo, un nuevo tipo de datos o nuevas funciones. Son útiles para reunir grupos de objetos en un solo paquete.

Para definir una extensión, se necesita por lo menos dos archivos, uno que sea un script que contiene los comandos SQL para crear objetos de la extensión, y un archivo de control que describe varias propiedades básicas de la extensión.

También pueden existir archivos con código en C si la extensión fue escrita en ese lenguaje [13]. Para agregarle a PostgreSQL una extensión es tan simple como agregarla al directorio extension y ejecutar el comando: `CREATE EXTESION nombre_de_extensión`.

Arquitectura de la extensión

La arquitectura de la extensión consiste en un esquema para cada base de datos documental y dentro de ellos funciones en PostgreSQL escritas en plpython, las cuales van a utilizar las bibliotecas de acceso a las distintas bases de datos NoSQL: pymongo para MongoDB y python-couchDB CouchDB.



Plpython

Plpython es un lenguaje procedural interno de PostgreSQL que permite que se escriba código de funciones en Python y sea ejecutado desde PostgreSQL. Es catalogado de desconfianza pues con él se puede acceder a recursos del servidor, pero si se utiliza correctamente se pueden lograr varias tareas, pues como puede escribirse código en Python se logra utilizar sus funciones y bibliotecas, las cuales están excelentemente documentadas. El lenguaje permite además varias funcionalidades propias, como acceso directo a ejecutar sentencias SQL y compartir variables globales entre sus diferentes funciones. Las funciones en plpython se ejecutan como si fuesen una función en SQL o Plpgsql [14].

Funcionalidades de las extensiones

Entre las funcionalidades de las extensiones están las siguientes:

- Gestión de una Base de datos (creación, eliminación, obtención).
- Gestión de documentos (creación, eliminación, modificación y obtención).
- Ejecución de funciones Map/Reduce.
- Obtención de la versión del servidor.

Además, cada una posee varias funcionalidades específicas del gestor como son:

CouchDB

- Compactación de documentos.
- Configuración de la replicación.
- Consultar una vista.
- Obtener la versión de un documento.
- Obtener un valor UUDI.

MongoDB

- Gestión de colecciones (creación, eliminación y obtención).
- Creación de índices.

A continuación se muestra una relación de las funciones de cada extensión:

MongoDB

- `collectiondelete`: Eliminar documentos de una colección.
- `collectionfind`: Buscar documentos en una colección.
- `collectionfindone`: Buscar un documento específico en una colección.

- `collectioninsert`: Insertar un documento en una colección.
- `collectionsimpleindex`: Crear un índice en una colección.
- `collectionupdate`: Actualizar un documento específico en una colección.
- `createcolleccion`: Crear una colección.
- `createdb`: Crear una base de datos.
- `deletecolleccion`: Eliminar una colección.
- `deletedb`: Eliminar una base de datos.
- `getcolleccions`: Obtener las colecciones.
- `getdatabases`: Obtener las bases de datos.

CouchDB

- `add_doc`: Agregar un documento.
- `createdbt`: Crear una base de datos.
- `db_compact`: Comprimir una base de datos.
- `db_del_doc`: Eliminar un documento.
- `db_get_doc`: Obtener un documento.
- `deletedb`: Eliminar una base de datos.
- `get_doc_rev`: Versión de un documento.
- `get_query`: Ejecutar Map/Reduce.
- `get_view`: Ejecutar vista.
- `upd_doc`: Actualizar un documento.

Creación y uso de las extensiones

Los archivos creados para las extensiones fueron los `*.control` y los `*.SQL`. A continuación un fragmento de los mismos:

Pgcouch.control

```
#pgpycouch extension
comment = 'extension para gestionar bases de datos couchdb desde postgresql'
default_version = '0.1'
relocatable = true
superuser = true
```

Pgmongo.control

```
#pgpymongo extension
comment = 'extension para gestionar bases de datos mongodb desde postgresql'
default_version = '0.1'
relocatable = true
superuser = true
```

Pgcouch.SQL

```
CREATE SCHEMA pgpycouch;
CREATE OR REPLACE FUNCTION pgpycouch.createdb(pserver text, pname text)
    RETURNS text AS $BODY$

import couchdb

#"http://ip:puerto"
```

```
servidor=couchdb.Server(str(pserver))

for bd in servidor:
  if bd==str(pname):
     return """El nombre de esa base de datos ya existe,
               por favor intente otro nombre!!!"""
return servidor.create(pname) $BODY$  LANGUAGE plpythonu VOLATILE;

CREATE OR REPLACE FUNCTION pgpycouch.add_doc(pserver text, pdb text, pdoc text)
    RETURNS text AS   $BODY$

import couchdb
import json

#"http://ip:puerto"
servidor=couchdb.Server(str(pserver))

bd=servidor[str(pdb)]
doc=json.loads(pdoc)

return bd.save(doc)

$BODY$
LANGUAGE plpythonu VOLATILE;
COMMENT ON FUNCTION pgpycouch.add_doc(text, text, text)
    IS 'agrega un documento nuevo a la BD el formato del doc es en texto'' "valor1"=1,"valor2"=2''';
```

Pgmongo.SQL

```
CREATE SCHEMA pgpymongo;
CREATE OR REPLACE FUNCTION
    pgpymongo.createdb(pserver text, pport integer, pname text)
    RETURNS text AS$BODY$

from pymongo import Connection

#servidor puerto
servidor=Connection(pserver,pport)
for bd in servidor.database_names():
  if bd == pname:
     return """El nombre de esa base de datos ya existe,
            por favor intente otro nombre!!!"""
servidor[pname].collection_names()
return 'Base de datos creada en: '+ str(servidor[pname])
$BODY$  LANGUAGE plpythonu VOLATILE;


CREATE OR REPLACE FUNCTION pgpymongo.collectioninsert(pserver text,
                                                      pport integer,
                                                      pbdname text,
                                                      pcollection text,
                                                      pdoc text)
    RETURNS text AS $BODY$

from pymongo import Connection

import json

#servidor puerto
```

```
servidor=Connection(pserver,pport)

bd =servidor[pbdname]
coll=bd[pcollection]
doc=json.loads(pdoc)
return coll.insert(doc)

$BODY$  LANGUAGE plpythonu VOLATILE;
```

Luego, para utilizar las funciones escritas en plpython se ejecutan con la sentencia *Select*

Ejemplos

Ejemplo de creación de una base de datos en couchdb

```
select pgpycouch.createdb('http://localhost:5984', 'nueva')
   Resultado- "<Database 'nueva'>"
```

Ejemplo de creación de un documento en una base de datos couchdb

```
select pgpycouch.add_doc('http://localhost:5984',nueva','
                        {"valor1":1,"valor2":"prueba"}').
       Resultado-"(u'96549d1e2...', u'1-4889fc1b...')"
```

Ejemplo de obtención de un documento en una base de datos couchdb

```
select pgpycouch.db_get_doc('http://localhost:5984', 'nueva', '96549d1e20dbc7c0d02c0eb6360003b2')
      Resultado- "<Document u'96549d...'@u'1-8c37b...' {u'valor1': 1, u'valor2': u'prueba' }>"
```

Ejemplo de creación de una base de datos en Mongodb

```
select pgpymongo.createdb('localhost', 27017, 'nueva')
      Resultado- "Base de datos creada en - Database(Connection('localhost', 27017), u'nueva')"
```

Ejemplo de creación de un documento en una base de datos Mongodb

```
select pgpymongo.collectioninsert('localhost', 27017, 'nueva', 'micoleccion',
                                  '{"valor1":1,"valor2":"prueba"}')
     Resultado- "5027cbf32229710460000002"
```

Ejemplo de obtención de un documento en una base de datos mongodb

```
select pgpymongo.collectionfindone('localhost', 27017, 'nueva',
                                   'micoleccion','{"valor2":"prueba"}')
    Resultado-"{u'valor1': 1, u'valor2': u'prueba', u'_id': ObjectId('5027cbf...')}"
```

Conclusión

Se evidenció la capacidad de extensibilidad que tiene PostgreSQL, incluso en otros lenguajes diferentes al que fue creado -Python en este caso-. Se mostraron dos extensiones pgpymongo y pgpycouch con funciones en plpython para interactuar con las bases de datos NoSQL documentales MongoDB y CouchDB respectivamente, presentando las funciones de cada una y con ejemplos que demuestran su utilización.

10	NoSQL and Document-Oriented Databases, 2010,Guy Harrison
11	Why Large Hadron Collider Scientists are Using CouchDB, 2010, Disponible: http://www.readwriteweb.com/enterprise/2010/08/lhc-couchdb.php.

12 Production Deployments, 2102, Disponible: http://www.mongodb.org/display/DOCS/Production+Deployments.

13 Packaging Related Objects into an Extension, PostgreSQL 9.1.0 Documentation 2012, Disponible: http://www.postgresql.org/docs/

14 PL/Python - Python Procedural Language, PostgreSQL 9.1.0 Documentation 2012, Disponible: http://www.postgresql.org/docs/

Prymatex

Autores: Nahuel Defossé, Diego van Haaster

Bio: Diego es Licenciado en Informática y trabaja con desde 2007 con Python, entre otros lenguajes. Es desarrollador independiente con Django pero con tendencia al front end rico. Fiel defensor de Gentoo. Actualmente intenta acercar el software libre a su lugar de trabajo desde el lado de las bases NoSQL.

Nahuel es Licenciado en Informática, docente e investigador en la UNPSJB. Trabja con Python desde el 2007 y lo enseña desde el 2008. Su background electrónico lo acerca al trabajo con bits en proyectos de control autónomo mechados con interfases en HTML y SVG.

Web: http://prymatex.org/blog

Emails: diegomvh@gmail.com nahuel.defosse@gmail.com

Twitter: @diegomvh @D3f0

Prymatex es un editor de texto multiplataforma basado en el popular TextMate de Mac. Al igual que éste, se compone de un editor minimalista extensible, y cuenta con una gran cantidad de extensiones que le brindan funcionalidad específica para muchos lenguajes y herramientas.

Cada extensión se denomina Bundle y puede contener varios recursos como:

- sintaxis
- comandos (compilar y usar herramientas externas, entre otras)
- snippets (trozos de códigos autoinsertables)
- macros (acciones pre grabadas)
- plantillas de archivos y proyectos
- preferencias de configuración

Cada componente de un bundle está contenido en un formato de serialización XML de Apple, llamado *plist* y que desde Python 2.6 se puede leer como un diccionario.

En el siguiente fragmento corresponde al plist del comando **Documentation in Browser** del bundle **Python**.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
 "http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>beforeRunningCommand</key>
    <string>nop</string>
    <key>command</key>
```

```
    <string>TPY=${TM_PYTHON:-python}

echo '&lt;html&gt;&lt;body&gt;'
"$TPY" "${TM_BUNDLE_SUPPORT}/browse_pydocs.py"
echo '&lt;/body&gt;&lt;/html&gt;'</string>
    <key>input</key>
    <string>none</string>
    <key>keyEquivalent</key>
    <string>^H</string>
    <key>name</key>
    <string>Documentation in Browser</string>
    <key>output</key>
    <string>showAsHTML</string>
    <key>scope</key>
    <string>source.python</string>
    <key>uuid</key>
    <string>095E8342-FAED-4B95-A229-E245B0B601A7</string>
</dict>
</plist>
```

El script que ejecutará Prymatex se encuentra en la clave **command** y, como se puede observar, consiste en un script escrito en Bash. El contexto de ejecución del script está definido por el resto de las claves. Al pulsar las teclas **keyEquivalent** se toma la entrada **input** para posteriormente ejecutar el script y dirigir la salida a **output**.

Si bien este comando está escrito en Bash, los scripts se pueden escribir en cualquier lenguaje que soporte leer y escribir la entrada estándar, interactuar con las variables de ambiente y retornar un código de salida entero. Debido a estos requerimientos mínimos, encontramos bundles con comandos escritos en Ruby, Bash, Perl, PHP, Python, awk, entre otros. Todo está limitado a lo que el usuario quiera poner en el shebang.

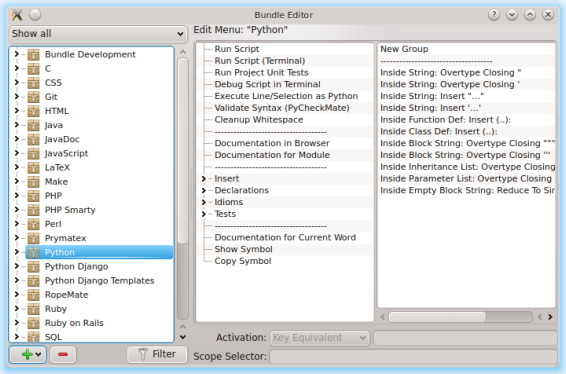
Esta forma sencilla de extender el editor ha dotado a TextMate de una gran popularidad. Una búsqueda en github con el texto *tmBundle* nos arroja más de 4000 repositorios como resultado. Muchos son forks de los bundles más populares, pero podemos decir sin dudas que TextMate es el editor que más lenguajes soporta.

Anatomía de un bundle

Analicemos con algo de detalle los componentes destacados de un bundle para comprender la capacidad de Prymatex como editor de textos para programadores o usuarios con necesidades de un editor extensible.

Los elementos que componen a un Bundle extienden la funcionalidad del editor significativamente, las sintaxis incorporan la metadata al texto tipeado y los comandos, snippets y macros definen las posibles acciones a ejecutar cuando se reúnen las condiciones adecuadas. Estas condiciones están dadas por un *ámbito*, *un atajo de teclado*, *la tecla de tabulación* o la combinación de varias.

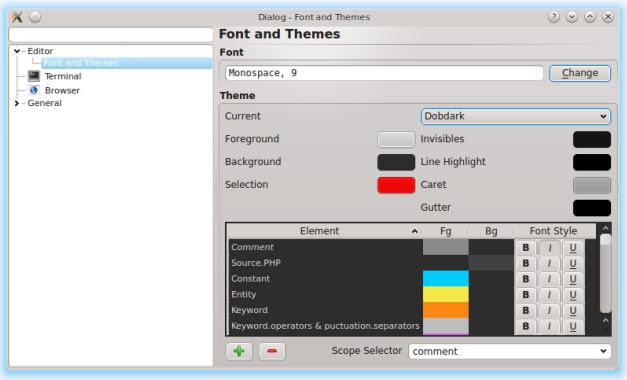
Todos los elementos que componen a un Bundle terminan siendo encapsulados en archivos *plist*. Estos pueden almacenarse en distintas ubicaciones dando lugar a los espacios de nombres.



Sintaxis

Los archivos de sintaxis definen principalmente la gramática del lenguaje.

Las gramáticas que se expresan en los archivos de sintaxis asocian cada palabra o símbolo analizado con un ámbito o *scope*. A medida que el usuario escribe en el editor, el resaltador de sintaxis asigna a cada carácter un *scope*. El *scope* es la *piedra angular* del editor y en base a él se definen muchas de las posteriores acciones (comandos, snippets, macros), como así también el típico coloreado de cualquier editor de texto para programadores.



Una gramática se define mediante expresiones regulares en Oniguruma, las cuales no son 100% compatibles con las del módulo nativo de Python, **re**. * Esta "no compatibilidad" se superó con *Ponyguruma*, un binding para Oniguruma desarrollado por Pocoo (autores de Flask, Jinja2, Pygments o Sphinx).

* Prymatex intenta utilizar *re* por razones de velocidad, pero si falla la compilación, recurre a Ponyguruma.

Comandos

Los comandos son acciones que pueden tomar datos del editor (documento, linea, carácter, etc.) y luego de ejecutar un script redirigir la salida nuevamente hacia el editor (insertar, remplazar, mostrar en el browser, etc).

Un comando que se repite en casi todos los Bundles es Run, y se ejecuta con la tecla Windows o Meta + R. La salida del comando generalmente se muestra en el browser integrado. Es destacable que no se necesita guardar incluso en lenguajes compilados como C o C++.

Snippets

Los snippets son pequeñas fracciones de texto que se utilizan para "alivianar" la inserción de código repetitivo.

Están definidos como texto, expresiones regulares y *"huecos"* o *holders*. Estos últimos representan los lugares variables a completar por el usuario y son navegables mediante la tecla de tabulación. Por ejemplo, bajo la sintaxis de Python, tras tipear *try* y presionar la tecla de tabulación, se inserta la definición de un bloque try/except y con cada tabulación el usuario puede modificar los holders definidos.

```
try:
    ${1:pass}
except ${2:Exception}, ${3:e}:
    ${4:raise $3}
finally:
    ${5:pass}
```

Proyectos

Prymatex provee un administrador de proyectos como un panel lateral que visualiza el contenido del sistema de archivos. Dentro de la carpeta del proyecto se genera un directorio oculto donde se almacena la meta información sobre el proyecto.

Los proyectos no solo sirven como organización lógica del espacio de trabajo sino que también definen en sí mismos un espacio de nombres; esto provee la posibilidad de generar Bundles dentro del proyecto y por lo tanto habilita a la redistribución para homogeneizar las tareas del grupo. En el menú contextual de un proyecto se pueden generar asociaciones con Bundles permitiendo ejecutar acciones sobre los archivos que contiene.

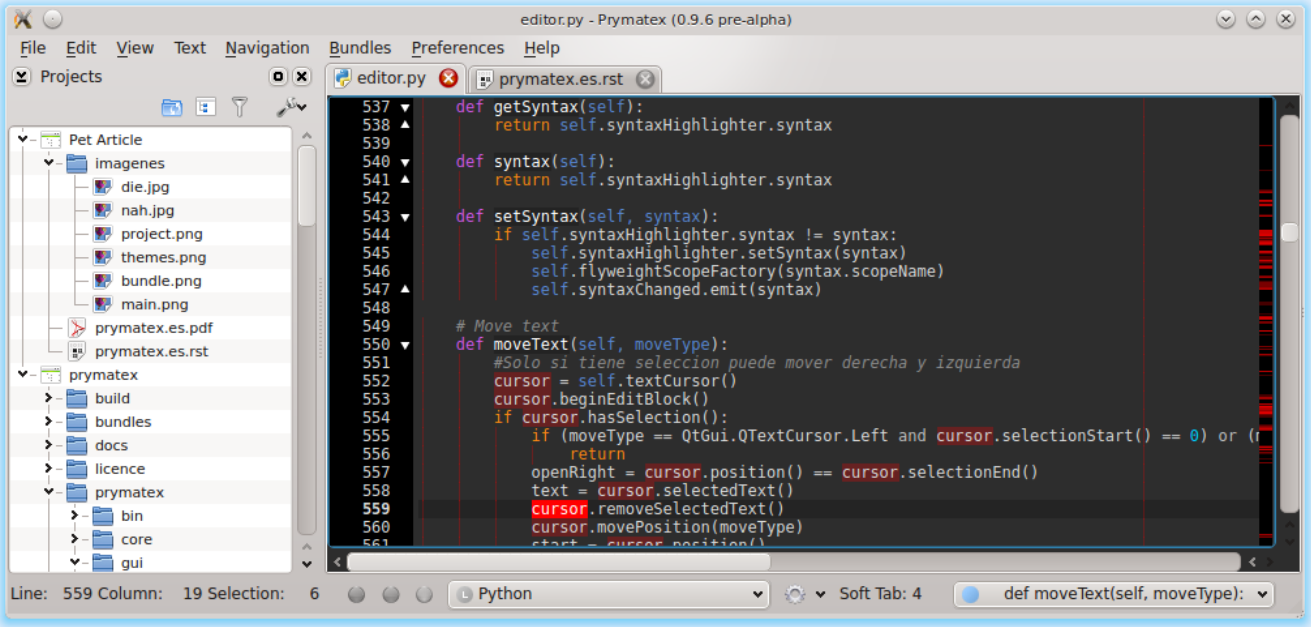
Pretendiendo extender las características de los Bundles de TextMate, Prymatex define plantillas de proyectos, de una forma similar a las plantillas de archivos. Éstas ayudan en la generación de código boilerplate que existe al iniciar un nuevo proyecto.

Edición

Prymatex incorpora varias herramientas para agilizar el tipeo de código. Buena parte de ellas están dadas por los macros o regidas por las preferencias de cada Bundle, y como tal se aplican según su *scope*.

El autocompletado básico del editor está basado en las palabras tipeadas y analizadas, aunque igualmente provee una API para hacer llegar al sistema de autocompletado sugerencias que ingresen de la ejecución de comandos.

Un aspecto que llama generalmente la atención es el modo multicursor: Prymatex activa este modo al seleccionar con el puntero zonas de código o mediante la pulsación de teclas específicas. Posteriormente con el modo activo podemos escribir en varios lugares del documento al mismo tiempo.



Instalación

Dependencias

```
$ sudo apt-get install python python-dev python-qt4 cmake git
$ sudo apt-get install x11-xserver-utils ipython python-zmq libonig-dev
```

Ponyguruma

```
$ git clone https://github.com/prymatex/ponyguruma.git
$ cd ponyguruma
$ python setup.py build
$ sudo python setup.py install
```

QTermWidget

```
$ git clone https://github.com/prymatex/qtermwidget.git
$ cd qtermwidget
$ cmake .
$ make
$ sudo make install
$ cd pyqt4
$ python config.py
$ make
$ sudo make install
```

Prymatex (sources)

```
$ git clone https://github.com/prymatex/prymatex.git
$ cd prymatex/prymatex/bin/
$ python pmx.py
```

Pyodel

Autor: Alan Etkin

Bio: Técnico de PC y redes LAN, programador. Estudié Diseño en la UBA y actualmente curso el profesorado de Matemática.

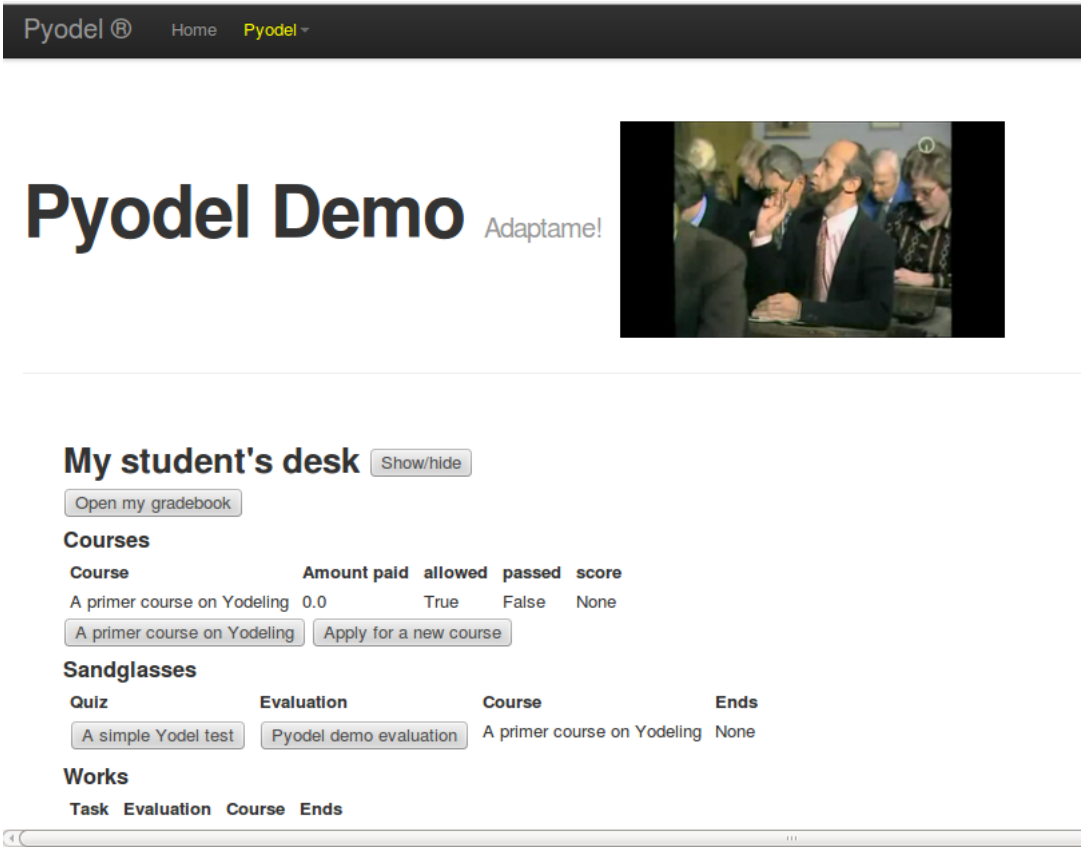
Web: http://spametki.blogspot.com

Email: spametki@gmail.com

Twitter: @spametki

A partir de la impúdica decisión de mi cliente de armar su propia aplicación Moodle para un emprendimiento de *e-learning*, y disponiendo de algún tiempo extra para pensar en algo más, inicié la búsqueda de una alternativa sobre **web2py** [15] para presentar una aplicación optimizada, que funcionara como *plugin*, de fácil mantenimiento y pytónica para el sofware de enseñanza en red.

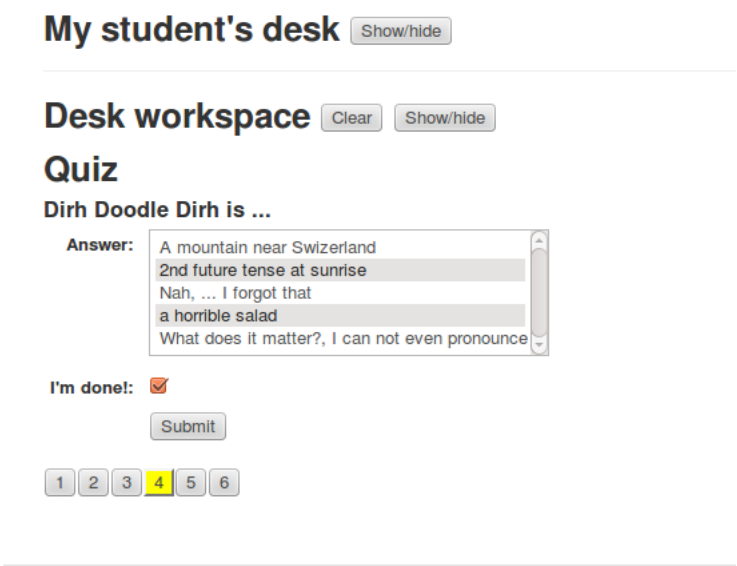
La propuesta consiste en la producción y desarrollo de un conjunto de herramientas para educación que se puedan utilizar como plugin en otras *apps* de web2py (a utilizarse por ejemplo en alguno de los actuales *CMS* disponibles para esa plataforma) o como una aplicación autónoma, es decir, un sencillo sistema integrado para enseñanza *online* y administración de pequeños centros de estudio.



¡Ahora disponible en Appspot!

- Modo plugin (todas las funcionalidades se pueden desplegar en cada aplicación por componentes individuales).
- Admisión a cursos y ABM de asistentes.
- Manejo y exposición de lecciones para estudiantes y docentes.
- Libretas de calificaciones con actualización asíncrona, soporte para cálculo de notas con abreviaciones de instancias de calificación y *permisología* de escritura y lectura (utilizando el módulo spreadsheet, de M. Di Pierro).
- ABM de calificaciones y resultados de exámenes por estudiante, curso o instancia.

- Edición y presentación de documentos educativos con la nueva característica incorporada auth.wiki de web2py y la sintaxis markmin.
- Servicio de streaming y manejo de cursos y lecciones con acceso restringido por asistente.
- Interfaz para admisión automatizada con servicios de pago electrónico
- Paneles personalizados para estudiantes o docentes.
- Administración y presentación de trabajos de estudiantes, exámenes y *quiz* (especie de trivia con opción múltiple o simple).
- Sintaxis simplificada para ingreso de quiz.



La documentación (en construcción) se puede consultar en la **wiki** del proyecto [16].

Para explorar las características desarrolladas, se debe ingresar a la **demo** [17] de Pyodel en appspot. Registrarse es obligatorio y gratuito. Incluye un curso de ejemplo, con algunos documentos, lecciones, stream de video y un quiz

No, no... Es Pyodel (pai-ö-dl)

Pyodel es una app *multipropósito* para web2py que permite ABM para e-learning con **RBAC** según administrador, docente y estudiante, crear y realizar exámenes, editar y consultar calificaciónes, dar soporte para streaming de lecciones con control de acceso y compartir documentación editada con la sintaxis **markmin** [18]. Está pensada como un simple sistema para centros educativos o dictado de cursos, funcionando en combinación con aplicaciones como **Instant Press** [19] o **Movuca** [20], en modo plugin, o en el formato de app autónoma. Se puede alternar el modo de desplegar Pyodel según los requerimientos específicos: sólo hay que descargar el instalador adecuado desde la página del **repositorio** [21] y luego cargarlo en la instalación local de web2py desde la interfaz administrativa.

El nombre está inspirado en un sketch de *Loriot* sobre el *Instituto para la enseñanza del Yodel* [22]

Requisitos

Para poder utilizar la implementación actual de Pyodel es necesario instalar previamente web2py. La instalación de web2py es verdaderamente sencilla. Las instrucciones y el paquete de descarga están disponibles en **web2py.com** [23]

¿Cómo se instala?

Instalando el plugin

Vamos a tener instalado Pyodel antes de que puedan pronunciar Holleree correctamente (si no pueden, no se preocupen). Los pasos son:

- Descargar el plugin en nuestro sistema desde la página de Pyodel en en **Google Code** [24]
- Abrir la interfaz administrativa de web2py y acceder al panel administrativo de la app específica en la que estamos trabajando (clic en el botón editar debajo del nombre de la app en la página **Sites**)
- En la sección inferior de la página, clic en el campo para subir un archivo (**plugin**) y especificar la ruta en el sistema al archivo **.w2p** descargado
- Listo. Ya tenemos instalado Pyodel en el ámbito de nuestra aplicación.

Configuración

Esto es aún más fácil. Por ahora, el plugin no tiene comandos de configuración inicial. Sólo debemos ingresar a la acción de configuración por única vez para que Pyodel cargue los registros de la demo en la base de datos. Se debe utilizar la siguiente url (reemplazando <app> por la url real de la aplicación) en el navegador para correr la configuración inicial.

```
http://<app>/plugin_pyodel/setup
```

De esta forma es posible realizar pruebas de los distintos componentes utilizando los registros de la demo como cursos, lecciones, documentos, stream o quiz.

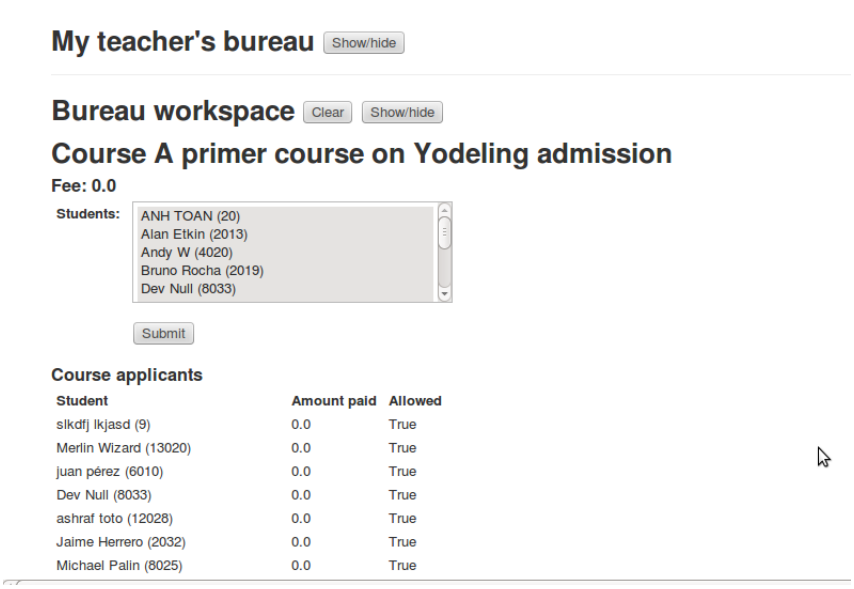
Componentes de plugin_pyodel

Luego de instalar Pyodel como plugin, podemos cargar los distintos componentes en cualquier vista de nuestra app con el ayudante **LOAD** de web2py.

```
...
{{=H3("Doodle")
{{=LOAD(c="plugin_pyodel", f="<componente>", args=[...], ajax=True)}}
...
```

Algunos de los componentes disponibles actualmente				
grid	sandglass	gradebook	grade	admission
bureau	desk	lecture	course	stream
hourglass	attendance	evaluation	wiki	

Por ejemplo, el *widget* de admisión a cursos que se muestra abajo corresponde al componente **admission**



Para los componentes asociados a un registro particular de la base de datos, se deben agregar los argumentos de tabla e id de registro, por ejemplo:

```
...
{{=H3("Holleree")
{{=LOAD(c="plugin_pyodel", f="lecture",
    args=["plugin_pyodel_lecture", 1],
    ajax=True)}}
...
```

La lista de tablas (incluyendo las tablas de Pyodel) se puede visualizar en la interfaz de ABM por defecto para la app, **appadmin** en

```
<url de app>/appadmin
```

o con la API de **DAL**

```
>>>>print [table for table in db.tables() if "pyodel" in table]
[..., "plugin_pyodel_course", ...]
```

Cómo utilizar la demo

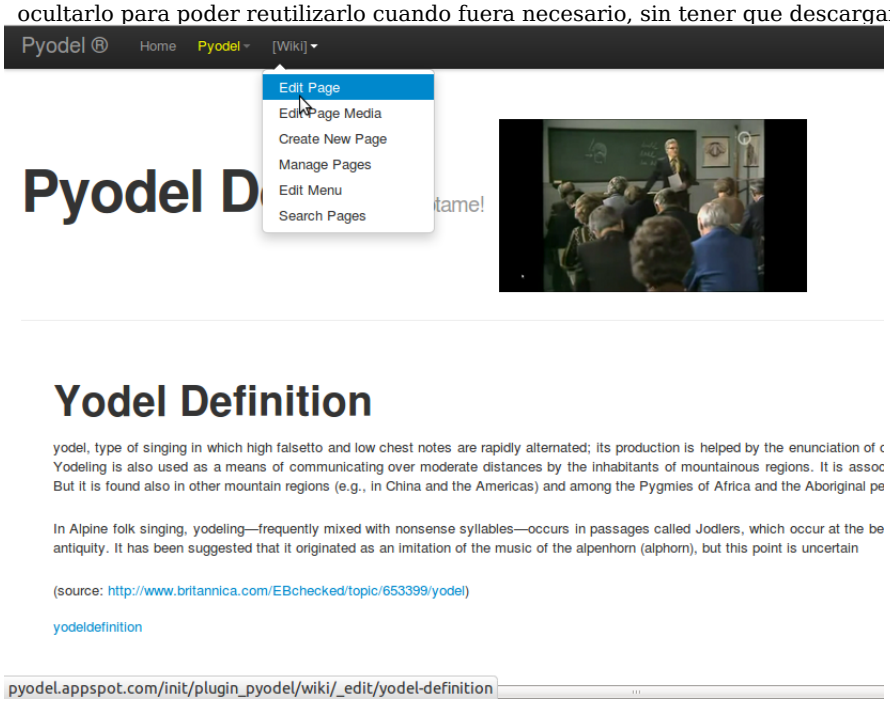
Primer paso: registrarse (es gratis)

Para comenzar a usar la demo en línea, necesitamos una cuenta de usuario registrado (como en casi toda app de web2py)

- Ingresar a la página de la demo en **pyodel.appspot.com**
- Ir al link registro
- Completar el simple formulario
- Eso es todo. Tenemos una nueva cuenta de estudiante en Pyodel

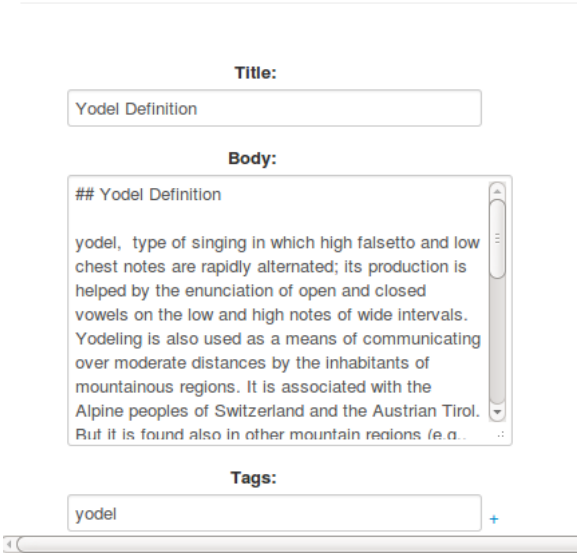
Interfaz para estudiantes

- Clic en el link **student desk** de la página inicial de la demo. La app debería presentarnos una serie de botones para acceder a los distintos recursos de la demo.
- Si hacemos clic en el curso **Yodel**, tendremos acceso a una descripción del curso, con una lista de videos disponibles, lecciones y documentos.
- Al hacer clic en cada botón se mostrará el recurso en un área llamada **workspace**, que está en algún lugar de la sección intermedia del cuerpo del documento web. Si se quiere, es posible deshacerse de los recursos en la página utilizando el botón **clear** o **show/hide** asociado. El uso de **clear** para un recurso determinado lo removerá del documento, mientras que usar **show/hide** implica únicamente ocultarlo para poder reutilizarlo cuando fuera necesario, sin tener que descargarlo nuevamente.



Por defecto, cuando hacemos clic en un recurso para visualizarlo, algunos elementos de la página se ocultan automáticamente. De esta forma se logra un poco más de economía del espacio en el documento y podemos concentrarnos en el último material consultado.

Cuando se selecciona un documento de la lista, se abandona la página actual para abrir la interfaz incorporada **wiki** de web2py, que permite realizar edición y visualización de todos los recursos accesibles para nuestro usuario, además del documento seleccionado. Hay que tener en cuenta que cada usuario puede agregar y modificar sus propios documentos wiki (utilizando la barra de menú superior con comandos de la utilidad) y explorar el directorio de documentos de la app. Para regresar al escritorio del estudiante, hay que acceder a través del link en página de inicio



Pyodel es *open source* y las propuestas de modificación y diseño son bienvenidas. Basta con ingresar al sitio del proyecto e iniciar un issue con una solicitud de revisión o mejora. El código fuente está disponible para descarga y modificación según los términos de la licencia **Affero AGPLv3** [25]

Para informar errores o sugerir modificaciones también es aconsejable abrir un reporte en la lista de **issues** [26].

Además se pueden ingresar quiz sobre web2py u otro tema para que estén disponibles en el sitio en Google Appspot, por medio de un formulario en la página de la demo.

15 http://www.web2py.com. web2py es el armazón empresarial para desarrollo de aplicaciones web sobre Python con interfaz para múltiples sistemas de bases de datos (*PostgreSQL*, *SQLite*, *DataStore*, *MySQL*) y desplegable en prácticamente cualquier entorno (Apache, Nginx, AppEngine)

16 http://code.google.com/p/pyodel/wiki

17 http://pyodel.appspot.com

18 http://www.web2py.com/examples/static/markmin.html

19 http://code.google.com/p/instant-press/

20 http://movuca.org/

21 http://code.google.com/p/pyodel

22 http://youtu.be/lliHC7QSiG8

23 http://www.web2py.com/examples/default/download

24 http://code.google.com/p/pyodel/downloads

25 http://www.gnu.org/licenses/agpl-3.0.html

26 http://code.google.com/p/pyodel/issues

Presente y futuro del entorno de desarrollo Sugar

Autor: Gonzalo Odiard

Bio: Aunque relativamente nuevo en el mundo pythonista, he desarrollado soft por más de veinte años en variedad de lenguajes. Actualmente trabajo como responsable del desarrollo de actividades en el equipo de One Laptop per Child Association

Web: http://godiard.blogspot.com

Email: gonzalo@laptop.org

Esta nota no se trata acerca de una nueva librería o framework que nos cambiará la vida, sino acerca de cómo un proyecto usa tecnologías ya existentes para ayudar a otros a cambiar sus vidas, cómo esto nos cambia a nosotros y cómo en algunos aspectos estamos ayudando a cambiar las tecnologías que usamos.

Breve introducción a OLPC

El proyecto One Laptop per Child [1] nace con la iniciativa de proveer laptops de bajo costo para ayudar a mejorar la educación de los niños del mundo que más lo necesitan. En ese momento, finales del 2005, cuando las notebooks no salían menos de 1000 dólares estadounidenses (USD), se hace una apuesta fuerte: saldrían 100 USD. Finalmente no se llegó a ese precio, sino a poco menos de 200 USD, pero se sacudió a la industria, creando el segmento de las netbooks. La computadora creada, llamada XO, tiene muchas características aún no superadas, como una pantalla que se puede leer a la luz del sol, entradas de audio que se pueden usar para conectar sensores, o baterías que duran 5 años, además de un diseño a prueba de niños.

Sugar y SugarLabs

Una de las líneas fundamentales del proyecto OLPC, es que el software debe ser libre.

El software desarrollado, denominado Sugar, es un entorno de aprendizaje, que incluye las actividades (como llamamos a los programas) necesarios para escribir, pintar, navegar por Internet, leer y obtener libros, hacer música, programar en varios lenguajes y jugar. Un par de centenares más de actividades se pueden descargar de nuestro portal [2].

Luego del desarrollo inicial dentro del proyecto OLPC, se creó SugarLabs [3] como un proyecto para coordinar el desarrollo de Sugar, bajo la tutela de la Software Freedom Conservancy, con lo cual Sugar puede usarse en cualquier computadora.

Sugar y Python

Desde el primer momento, se decidió usar Python para el desarrollo de Sugar y las actividades. En línea con el uso de software libre, para posibilitar la apropiación y modificación por parte de niños, maestros y comunidades locales, el uso de Python con su simplicidad y potencia decidió al equipo de desarrollo, a pesar del costo en performance, en especial con hardware que tenía características por debajo de los de última generación. Sugar facilita ver el código fuente de las actividades y clonarlas para modificarlas sin correr riesgos.

Seymour Papert, creador del Logo y uno de los primeros en utilizar computadoras en educación, decía que el *"debugging es la esencia de la actividad intelectual"*. Analizar qué es lo que está mal en un programa, nos hace pensar acerca de los procedimientos, desmenuzarlos, probar distintas alternativas, y descubrir relaciones de causa y efecto. Es experimentar y crear.

Es por esto que aprender a programar no es un fin en sí mismo en el proyecto, sino un medio para desarrollar el pensamiento abstracto, la lógica, la creatividad, el trabajo en equipo, el descubrir que se pueden pensar distintas soluciones a los mismos problemas.

Aunque nunca fue un objetivo principal, hoy en día comenzamos a ver los frutos de esta filosofía llevada a la práctica. Más del 10% de las actividades en nuestro portal, han sido desarrolladas por niños que han crecido usando Sugar y hoy tienen menos de 15 años de edad. Algunos de ellos participan de nuestras reuniones virtuales de diseño y comienzan a mandar parches para las librerías del proyecto.

Presente y Futuro

Las XO y Sugar están siendo usadas por más de 2.4 millones de niños en más de 40 países; en algunos lugares por todos los alumnos y maestros del país, como en Uruguay y Perú, en otros lados en una provincia, como en La Rioja, Argentina. En otros lugares en una ciudad, poblado o escuela. Sugar también se incluye en distintas distribuciones Linux, y se puede usar desde un CD o un pendrive en cualquier computadora.

Luego de la versión inicial de nuestra computadora, la XO-1 [4], que entró en producción en noviembre del 2007, le siguió la XO-1.5 [5], en el 2009 y en el 2011, para bajar aún más el consumo eléctrico, un elemento muy importante en algunas comunidades apartadas, se desarrolló la XO-1.75 [6], usando un procesador ARM.

En este momento nos hallamos desarrollando el hardware y software de la XO-4 [7], que además de aumentar las prestaciones del procesador, memoria y almacenamiento, por primera vez va a contar con la posibilidad de incluir una pantalla táctil.

Esto nos ha hecho tener que re-analizar nuestra interfaz con el usuario, ya que por ejemplo no se cuenta con un "botón secundario" cuando se usa un dedo, o eventos asociados a posicionar un cursor sobre un elemento, sino que hemos tenido que mejorar la interacción con teclados en pantalla y modos de selección de texto. Estamos contribuyendo con estos desarrollos a nuestros proyectos "upstream", GTK, los bindings dinámicos de Python, WebKitGtk, Abiword y Evince entre otros. Nuestra comunidad de traductores, trabajando en más de 100 lenguajes, ayuda a traducir otros proyectos. El círculo virtuoso del software libre se cierra, nos ayuda llegar hasta el límite de lo que se puede hacer hoy en día, y una vez allí, podemos ayudar a otros a ir aún más adelante.

Cómo participar

La comunidad SugarLabs es sumamente abierta. Varios integrantes de PyAr participan activamente o han participado en el pasado, y en los últimos años, hemos hecho actividades conectadas a las PyCon y PyCamps. En este año 2012, un día completo, el 15 de noviembre, será dedicado al desarrollo de Sugar. También es posible solicitar hardware para el desarrollo de proyectos. Pueden comunicarse con el autor de esta nota, con nuestra lista de desarrollo [8] o nuestra lista en español [9]

Sugar es un proyecto que se alinea con una característica muy presente en la comunidad de PyAr: el interés por lo social y el bien común, las ganas de hacer algo para que otros tengan mejores oportunidades y el encarar desafíos intelectuales.

¡Los esperamos!

1 http://one.laptop.org/

2 http://activities.sugarlabs.org/en-US/sugar/

3 http://wiki.sugarlabs.org/go/Welcome_to_the_Sugar_Labs_wiki

4 http://wiki.laptop.org/go/Hardware_specifications

5 http://wiki.laptop.org/go/Especificaciones_Hardware_XO-1.5

6 http://wiki.laptop.org/go/XO-1.75

7 http://wiki.laptop.org/go/XO-4

8 http://lists.sugarlabs.org/listinfo/sugar-devel

9 http://lists.sugarlabs.org/listinfo/sugar-desarrollo

A los programas los determina el programador, ¿y al programador...?

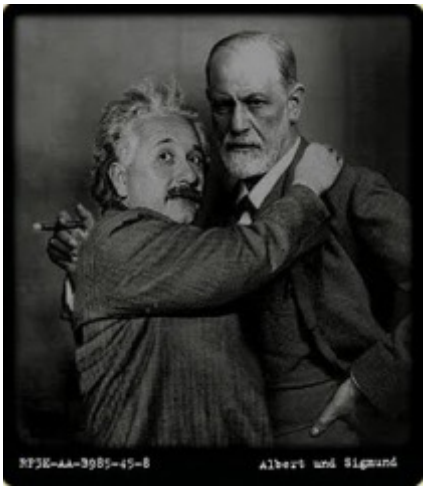
Autor: Javyer DerDerian

Bio: Desarrollador Web desde hace unos 10 años. 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



Einstein y Freud

Todos los que nos dedicamos – o nos hemos dedicado – al desarrollo de Software pasamos por la típica escena de dedicarle incontables horas, tardes... y hasta noches enteras a develar el secreto mejor guardado de una aplicación:

> ***¿POR QUÉ M....A NO FUNCIONA?*** *¡Si todo lo que tiene que estar bien está bien! ¡Los mensajes de error no tienen sentido y no hay lógica en este mundo que explique por qué está fallando lo que está fallando!*

Ninguno de los programadores que esté leyendo este artículo puede sentirse fuera de eso... ¡y si lo hace es porque todavía no trabajó en el desarrollo de software el tiempo suficiente!

Horas, tardes, noches, días, semanas... (¿años?), tratando de entender por qué todo lo que escribí de pronto falla... si todas las pruebas que hice estuvieron bien.

Hasta que de pronto, en un rapto de lucidez (o de "dormidez"), encontramos la maldita línea de código que explica – normalmente con un detalle totalmente simple y que no podemos creer que durante tanto tiempo se nos escapó – por qué fallaba lo que fallaba.

Pero esto mismo que nos pasa al programar también nos pasa a todos los que vivimos, y se repite en todas las áreas de la vida, con problemas que no distinguen si tienen código fuente y pantalla o son de carne y hueso.

Por lo que, afortunadamente, podemos dejar -tal vez por un tiempo- de pensar que es un problema que nos atañe solamente a los programadores, y recordar que es un problema de la condición humana.

Si hay una disciplina que puso en su centro el estudio de la condición humana, y el entramado de inhibiciones, síntomas y angustias que tratamos de sortear diariamente, es el Psicoanálisis.

Una disciplina con ya más de 100 años que, desde sus inicios y aún todavía, es vista por la sociedad en general como molesta, ya que en cada etapa se ocupa de interrogar exactamente lo que nadie hasta ese momento interroga.

Sigmund Freud, Fundador del Psicoanálisis, tuvo la valentía de decirle al mundo que los niños tienen sexualidad. Tomó el desafío de Escuchar los Sueños y Ver en Eso lo que el psiquismo estaba tratando por todas las vías de decir.

Lacan, el mayor exponente del Psicoanálisis luego de Freud, se atrevió a interrogar abiertamente la "verdad" política de que la felicidad de cada uno resultará de la felicidad de todos. La "verdad" económica de que primero hay que trabajar, y que el deseo viene después.

Lacan

Por eso, y por otras cuestiones más, desde siempre la ortodoxia científica trató por todas las vías de desacreditarlo, pero no por eso el Psicoanálisis se cayó.

Obviamente no pretendo ponerme a la altura de 2 de las personas más inteligentes del siglo XX… simplemente vengo a decir que luego de más de 10 años de dedicarme a la programación y 8 años de estudiar Psicoanálisis, poco a poco fui empezando a encontrar algo que al principio me parecía imposible:

un punto de encuentro entre 2 de los intereses más grandes en mi vida

El Psicoanálisis habla, principalmente, de hablar. *Lacan* puso la lupa principalmente en las estructuras discursivas, del "lenguaje".

Los programadores pasamos horas día a día con la lupa en las instrucciones, que a través de palabras escritas en un lenguaje que elegimos, le damos a las computadoras.

El Psicoanálisis se toma muy en serio demostrar que cada humano es "programado" por lo que, a través del "lenguaje", se le dice desde pequeño. Los programadores tenemos vasta experiencia en ver que con un lenguaje podemos programar algo, y que ese algo luego responderá en base a la programación sin intentar moverse siquiera un poco de eso que le programamos.

La realidad que percibimos no es "La Realidad", sino que está filtrada por todos los preconceptos que tenemos y que sólo la accedemos –y creamos- mediante el uso del lenguaje.

Los programadores no logramos acceder a la "realidad" de la computadora, sólo logramos, a través del lenguaje que previamente elegimos, acceder a lo que ese lenguaje nos permite hacer. Y lo que ese lenguaje no nos permite hacer nos queda totalmente imposibilitado.

Estos son sólo 3 puntos de articulación que intentan, y ojalá lo logren, mostrar que para quienes dedicamos horas y horas cada día a crear mediante el lenguaje, la Psicología es una de las disciplinas que más puede aportarnos para:

- Aprender a detectar las trampas (que el lenguaje nos crea).
- No caer en esas trampas (!)
- Aprovechar eso y ponerlo a favor de lo que queremos lograr; teniendo así la oportunidad de ser más eficaces en lo que hacemos para crear lo que queremos crear.

Para quienes hacemos de la sintaxis, la lógica, el álgebra y el uso del lenguaje nuestro día a día sería un desperdicio que sigamos tratando de avanzar sin resolver ni aprovechar lo que el Psicoanálisis nos enseña a resolver y aprovechar.

Fue *Albert Einstein* quien nos enseñó que es imposible resolver un problema desde el mismo nivel de pensamiento que lo generó.

El mismo *Albert Einstein* que ante la pregunta sobre la condición humana que nunca logró responder no tuvo mejor idea que preguntarle a Freud, en un intercambio epistolar histórico: ¿por qué guerra?

A lo que Freud respondió señalando la pulsión de muerte que es lo que en nuestro estado natural guía nuestros pensamientos inconscientes, mientras en la conciencia pensamos todo lo contrario. Pero la realidad de la humanidad, desde hace siglos, nos muestra que lo que termina determinando es lo inconsciente.

Años después *Lacan* diría, en su "Seminario 7" que

"Los procesos del pensamiento, sólo son conocidos a través de la palabra"

y es desde ese punto, reitero, quienes dedicamos largas horas todos los días a Crear con Palabras que tenemos la oportunidad de enterarnos las trampas que anidan en nuestros pensamientos, pero que sólo logramos detectarlas gracias a lo que surge en nuestras palabras. Palabras que no necesitan ser dichas, ya que las escribimos cotidianamente en "instrucciones", funciones, "argumentos" y clases.

La intención de este artículo es introducir un tema que nos permitirá, al profundizar en eso, desarrollar cada vez más articulaciones y aplicaciones del psicoanálisis en la programación –e informática–, día a día.

Quedan temas pendientes:

- ¿Cómo comprender cuál es el efecto del inconsciente Freudiano sobre el software que creamos?
- ¿Qué lógica podemos ver en nuestro software gracias a leerlo a través de los 4 Discursos desarrollados por *Lacan*?
- ¿Qué peso tienen los Significantes que usamos en nuestro software sobre la eficacia del mismo?
- ¿En qué consiste, y cómo podemos usar, la Lógica Peirceana en el planeamiento y desarrollo de una aplicación?

Varios interrogantes que Abren Caminos, y que tienen respuestas que son mundos en sí mismos.

Ojota

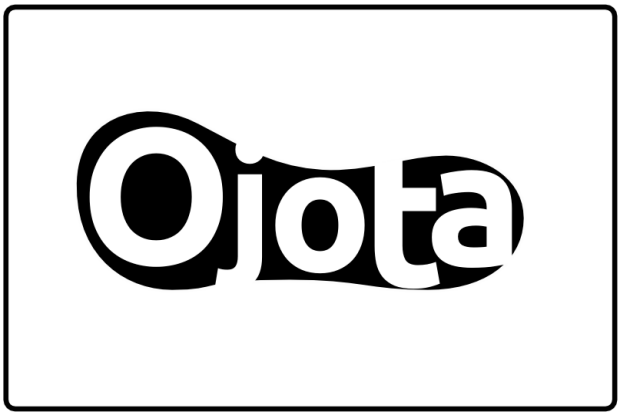
Autor: Felipe Lerena

Bio: Dressed by Mozilla. Powered by Milanga with fritas. First wave vaporecist. Playing with the interwebs. Hacking life, food, situations. Trolling you!

Web: http://www.felipelerena.com.ar

Email: felipelerena@gmail.com

Twitter: @felipelerena



En la empresa para la que trabajo, necesitábamos una base de datos cuya información esté lo suficientemente estructurada como para ser recuperada rápidamente, pero que no esté guardada en forma binaria, sino en texto claro para ser visualizada con un simple editor de texto. Por otra parte, las bases de datos “Flat File” tradicionales, como *SQLite* no nos servían porque almacenan la información en archivos binarios y hacen muchas más cosas de las que necesitábamos.

El código fue liberado y mejorado en pycamp 2012 (http://python.org.ar/pyar/PyCamp/2012) y fue creciendo cada vez más, agregando funcionalidad y robustez gracias a la colaboración de miembros de la comunidad de Python Argentina.

El requerimiento inicial fue que hiciera pocas cosas pero que las hiciera bien, rápido y de manera liviana. Desarrollamos una especie de motor de base de datos hecho 100% en Python que sabe leer y escribir a disco y que maneja la información en una Cache. El motor tiene integrado un ORM que nos permite hacer búsquedas y ordenar la información, así como cruzar información de varias fuentes de datos.

Cada clase tiene asociado una fuente de datos (que puede ser *JSON*, *YAML*, o un Web Service que devuelva *JSON*) y lee toda la información desde ahí y nos permite hacer consultas, ordenar, filtrar y hasta actualizar información y guardarla. Se comporta, en este sentido, como un *ORM* tradicional. También tiene soporte para distintos back-ends de Cache (en memoria, memcache o simplemente sin usar cache) lo que permite que se lea muy poco a disco.

Algunos ejemplos

```
# La información está en el archivo Persons.json, Por defecto se asume que
# la fuente de información es JSON
class Person(Ojota):
    required_fields = ("name", "address", "age")
    cache  = Memcache()

# La información está en el archivo Teams.yaml
class Team(Ojota):
    pk_field = "id"
    data_source = YAMLSource()
    required_fields = ("id", "name", "color")

    def __repr__(self):
        return self.name

# Algunos ejemplos de búsquedas:
# "all" devuelve una lista de objetos Person
Person.all(age=30, sorted="name")
Person.all(age__lt=30, sorted="-name")
Person.all(sorted="name")

# "get" devuelve un objeto Team, si encuentra más de uno devuelve el
# primero que encuentre
Team.get(1)
Team.get(name="River Plate")
```

Tanto las fuentes de datos como los motores de cache pueden ser extendidos para crear motores personalizados para la necesidad de cada usuario. Algo interesante que tiene es que cada fuente de datos (lo que en una DB tradicional llamarían “tabla”) puede ser de un tipo y un origen distinto.

Ojota también maneja las relaciones entre los objetos, permitiéndonos “seguir” las relaciones que tiene un objeto con otro, pudiendo traer todos los objetos que hacen referencia a éste, dándole a Ojota capacidades de base de datos relacional.

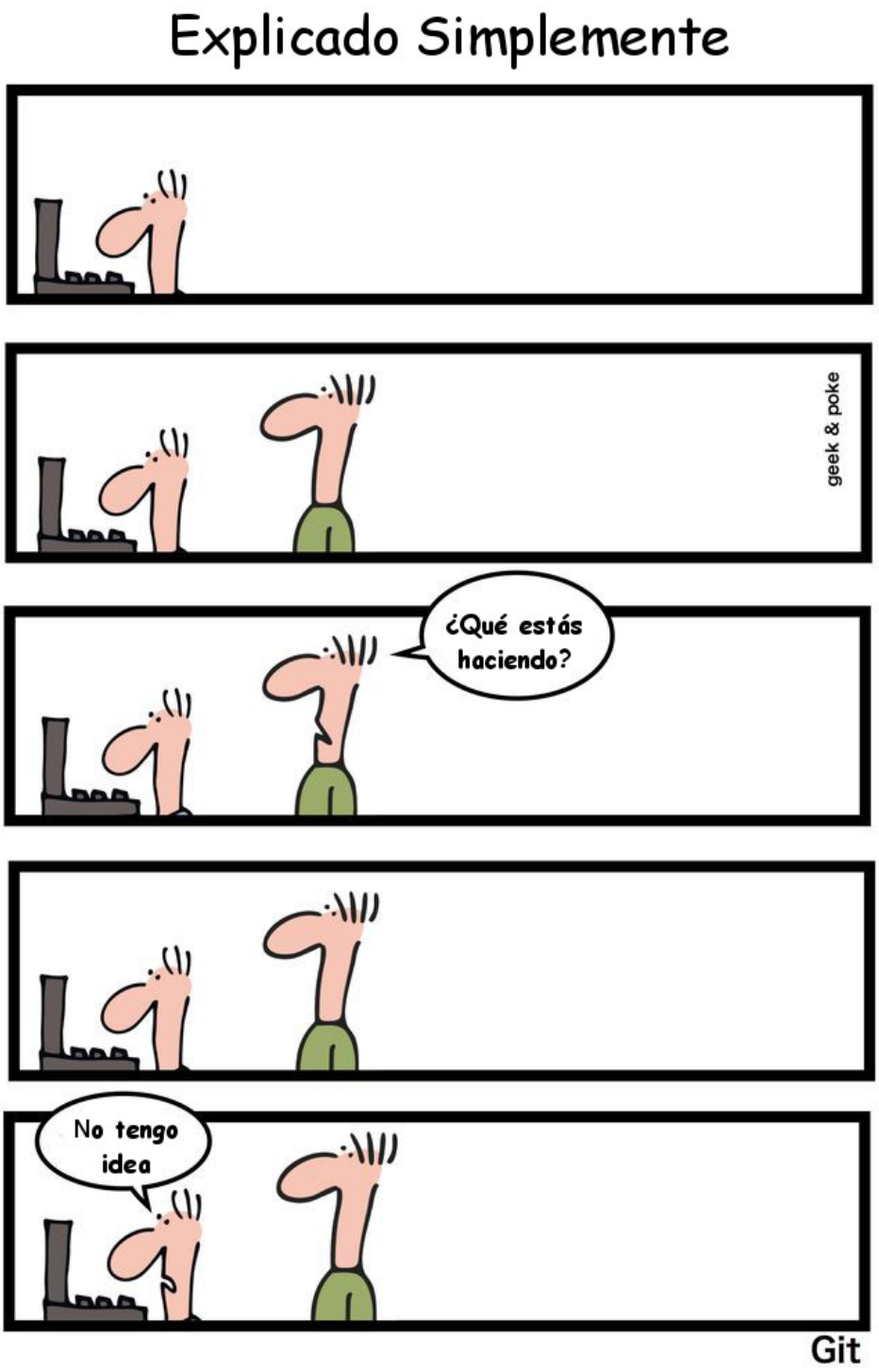
```
class Person(Ojota):
    pk_field = "id"
    required_fields = ("id", "name", "address", "age", "team_id")
    team = Relation("team_id", Team, "persons")
    country = Relation("country_id", Country, "persons")
    cache  = Memcache()

# Algunos ejemplos de relaciones
# Trae el pais de la persona
country = Person.get(1).country
country_name = country.name
# Trae todas las personas para un pais
persons_for_country = Country.get(1).persons
```

Pueden leer la documentación que está en http://ojota.readthedocs.org. Probarlo instalándolo desde Pypi http://pypi.python.org/pypi/Ojota o viendo directamente el código fuente en bitbucket https://bitbucket.org/msa_team/ojota

También estoy desarrollando una interfaz web para explorar Ojota que se llama **Havaiana**: pueden ver el código y algunos ejemplos en https://bitbucket.org/felipelerena/havaiana.

Geek & Poke



Este cómic proviene de Geek & Poke (http://geekandpoke.typepad.com)