# PET: Python entre Todos

Revista de la Comunidad Python de Argentina

#3
Jul 2011

## Especial:

## PyDay Córdoba 2011

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

ISSN: 1853-2071

# Licencia



# En Este Número




## Staff

**Editores:** Juan B Cabral - Tomas Zulberti. **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

# Y...Python cambió mi vida (Seguro vos me entendes)


**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

**Twitter:** @juanbcabral

Si bien al cierre del evento dije que mis compañeros solo fueron colaboradores, eso no es cierto, fue una palabra muy pequeña que lastimó bastante a algunos.

Este evento fue laburo de mucha gente, si bien metí el pecho como coordinador general, mucha otra gente hizo posible el evento:

Sin Santiago moreno, no hubiéramos tenido lugares, ni cañon, ni pizarrones, ni todo lo que dio la universidad.

Sin Emilio Ramirez, la comida no hubiera sido posible, todos tendrían que haber ido hasta el centro y volver en 1:30 para comer, así como el asado de la noche (con ping pong y metegol) no hubiera existido

Sin Tsunami Boom (Anggie), probablemente no hubiéramos llegado a las 100 personas inscriptas, con lo cual los asistentes hubieran sido no más de 30.

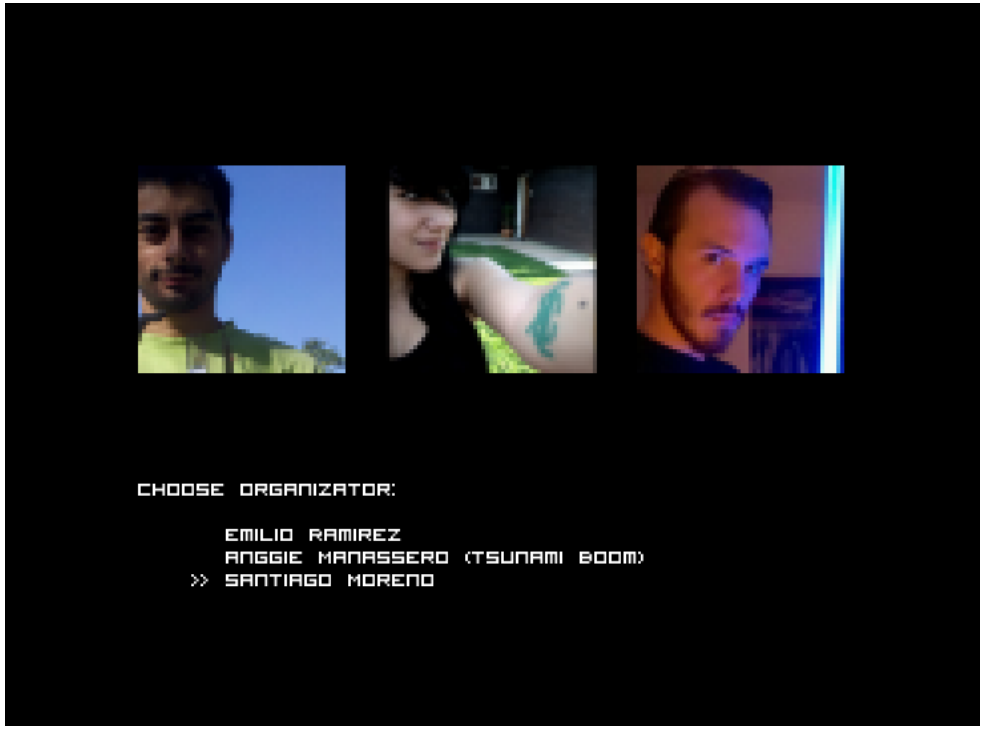
Además de ellos... mucha otra gente ayudo en el momento de la organización sin que se le pida mucho: Catriel Muller, Nestor Navarro, X-Ip, JJConti, Tomás Zulberti, Facu Batista... y casi todos los que estuvieron ahí.

Este evento, en definitiva, perteneció a todos los que fueron y a los que no fueron. Te pertenece a vos, que lo pudiste apreciar por una llamada telefónica, un tweet, algun post, unas fotos o esta revista... a vos miembro de Pyar que cambiaste mi vida, le dedico esta revista por que te lo mereces.

Fue un gusto...

**Juan B Cabral** Coordinador General - Pyday Cordoba 2011 Editor Pet Nro.3

**Los Organizadores:**



# Python en tu filesystem: Creando filesystems en userspace con Python y Fuse


**Autor:** Roger Durán
**Bio:** Linuxero, pythonista y con problemas sociopáticos
**Email:** roger@elvex.org.ar
**Twitter:** @roger_duran
**Web:** http://www.fsck.com.ar

## Primero que nada veamos que es un filesystem.

Una forma simple de verlo, seria un método de guardar datos y organizarlos de una forma que sea fácil encontrarlos y accederlos, estos se representan como directorios y archivos, estos usualmente se almacenan en dispositivos como puede ser un disco rígido o una memoria.

**Algunos ejemplos conocidos:**

- ext2 (http://en.wikipedia.org/wiki/Ext4)
- ext3 (http://en.wikipedia.org/wiki/Ext3)
- ext4 (http://en.wikipedia.org/wiki/Ext4)
- reiserfs (http://en.wikipedia.org/wiki/Reiserfs)
- fat (http://en.wikipedia.org/wiki/File_Allocation_Table)
- ntfs (http://en.wikipedia.org/wiki/Ntfs)

## ¿Que es Fuse?


Fuse (http://fuse.sourceforge.net/) es un modulo de kernel, como podría ser cualquiera de los nombrados anteriormente, pero que en vez de implementar un filesystem, nos expone su API en espacio de usuario. Pero... ¿Qué sistemas operativos puedo usar fuse?

- En la mayor parte de los sistemas operativos *nix, como puede ser GNU/Linux, MacOS, *bsd..
- En Windows existe una implementación de fuse llamada “Dokan”, no puedo comentar mucho sobre esta, ya que no la he probado.

Algunos ejemplos de uso de fuse

- **En Python:**
  - fusql
  - Gmailfs
  - Youtubefs
  - Cuevanafs
  - FatFuse
- **Otros lenguajes:**
  - gnomeVFS2
  - zfs-fuse
  - sshfs
  - etc..

Como podemos ver, podemos crear desde un filesystem mapeando recursos de internet/red, hasta filesystem tradicionales como puede ser zfs(muy popular en solaris) o fat.

**Fusql** es un fs fuera de lo común, lo que lo hace interesante es que mapea una base de datos relacional(**Sqlite**) como si fuera un filesystem, pudiéndose operar sobre ella completamente.

Las ventajas que tenemos al desarrollar un filesystem con Fuse son:

- Podemos programar en nuestro lenguaje favorito (en este caso python).
- Simplemente reiniciando la aplicación, estaremos haciendo pruebas con el codigo actualizado.

- Podemos acceder a las librerías del sistema para crearlos, por ej el stdlib de python.
- No tendremos que lidear con kernel panics, reinicios, utilización de maquinas virtuales para las pruebas, etc.
- Mayor portabilidad, ya que fuse existe en diversos sistemas operativos.
- Podemos ejecutar nuestros filesystem con permisos de usuario.
- Facil debugging de los mismos.

## Fuse: El API

El API de fuse funciona por callbacks. Por ejemplo: cuando nosotros accedemos a un directorio, en nuestra aplicación se llama a getattr, opendir, readdir, releasedir.

```
create(path, mode) # creación de un archivo
truncate(path, mode) # achicar o agrandar el tamaño de un archivo
open(path, mode) # apertura de un archivoError: BadDrawable
write(path, data, offset) # escritura de un archivo
read(data, lenght, offset) # lectura de un archivo
release(path) # liberación de un archivo
fsync(path) # sincronización de un archivo
chmod(path, mode) # cambio de permisos
chown(path, uid, gid) # cambio de propietario
mkdir(path, mode) # creación de directorio
unlink(path) # eliminación de un archivolink
rmdir(path) # eliminacón de un directorio
rename(opath, npath) # renombrado
link(srcpath, dstpath) # creación de un link
```

**Como se usa**

Este es un ejemplo mínimo de lectura y escritura de un archivo, hagamos de cuenta que estos métodos están en un objeto que tiene un diccionario llamado items con el path como key y los datos de este como valor

```
# lectura
def read(self, path, offset, length):
    # establecemos el comienzo de nuestra lectura
    start - offset

    # establecemos el fin de la lectura
    end - start + length

    # retornamos la cantidad de datos solicitado
    return self.items[path][start:end]

# escritura
def write(self, path, offset, data):
    # tomamos el tamaño de los datos a escribir
    length - len(data)

    # tomamos los datos actuales de nuestro archivo
    item_data - self.items[path]

    # agregamos/reemplazamos la parte del archivo que se solicito
    item_data - itdat[:offset] + data + item[offset+length:]

    # remplazamos el contenido de nuestro item
    self.items[path] - item_data

    # devolvemos el tamaño de datos que escribimos
    return length

# truncate
def truncate(self, path, length):
    # tomamos los datos de nuestro archivo
    item_data - self.items[path]

    If len(item_data) > length:
        # si el tamaño de nuestros datos es mayor que el solicitado
        # lo acortamos a este
        self.items[path] - item_data[:length]
    else:
        # sino rellenamos con 0 al tamaño solicitado
        self.items[path] +- '0' * len(item_data)
```

## Defuse

Trabajando con python-fuse, una de las cosas que me parecieron incomodas es el manejo de los path, viniendo del mundo de las webs, sobre todo con werkzeug/flask, se me ocurrió implementar algo similar a su manejo de rutas, pero para la escritura de un filesystem, así es como nació defuse https://github.com/Roger/defuse.

Lo que esto nos permite es utilizar decoradores para el manejo de las rutas, separando cada parte de nuestro filesystem como una clase con todos los métodos que nos proporciona fuse.

**Un pequeño ejemplo**

```
fs - FS.get()

@fs.route('/')
class Root(object):
    def __init__(self):
        root_mode - S_IRUSR|S_IXUSR|S_IWUSR|S_IRGRP|S_IXGRP|S_IXOTH|S_IROTH
        self.dir_metadata - BaseMetadata(root_mode, True)

    def getattr(self, *args):
        return self.dir_metadata

    def readdir(self, *args):
        for i in xrange(4):
            yield fuse.Direntry('test%s.txt' % i)


@fs.route('/<filename>.<ext>')
class Files(object):
    def __init__(self):
        file_mode - S_IRUSR|S_IWUSR|S_IRGRP|S_IROTH
        self.file_metadata - BaseMetadata(file_mode, False)

    def getattr(self, filename, ext):
        self.file_metadata.st_size - len(filename*4)
        return self.file_metadata

    def read(self, size, offset, filename, ext):
        data - filename * 4
        return data[offset:size+offset]
```

En el ejemplo anterior podemos ver una implementación funcional de un filesystem que muestra 4 archivos, cuyo contenido es el nombre del archivo repetido otras 4 veces.

Como pueden ver el manejo de los path se hace mediante el uso de decoradores de clase, además de que en cada método, ya no nos llega el path, sino que se agrega a estos las variables que definimos en el decorador.

Por ej: `@fs.route('/<dir1>/<dir2>/<archivo>.<ext>')` en `/root//subdir/test.py` nos devolvería las variables:

- `dir1-'root'`
- `dir2-'subdir'`
- `archivo-'test'`
- `ext-'py'`

## Conclusión

Este articulo mas que enseñarles todo acerca de python-fuse, lo que quiere es mostrarles lo simple que es utilizarlo y entusiasmarlos a que escriban sus propios filesystems.

En el pyday hubo algunas ideas interesantes, como traductores automáticos de archivos, o análisis con nltk.

Espero ver pronto sus filesystems!

# Desarrollo de Plugins en python para software educativo TurtleArt


**Autor:** Valentín Basel

**Bio:** Soy analistas en sistemas informáticos y me desempeño como encargado del área informática del CEA-CONICET. Comencé en el mundo de GNU/LINUX por el año 2000 cuando instale un SUSE 6.0 y decidí dedicarme al mundo del Software libre. Actualmente soy embajador Fedora por Argentina y mi aporte para la comunidad GNU es sobre robótica educativa libre con el proyecto ICARO, un entorno de software y hardware de bajo costo para el desarrollo de robots educativos en colegios primarios y secundarios.

Mi paso por python es relativamente nuevo (un par de años) anteriormente siempre programe en C/C++, sin embargo la facilidad para crear código, la excelente documentación y demás comodidades hicieron que cada ves mas programe en python como lenguaje principal.

**Email:** valentinbasel@gmail.com

**Web:**

- http://www.sistema-icaro.blogspot.com/
- http://valentinbasel.fedorapeople.org/

TurtleArt (http://wiki.laptop.org/go/Turtle_Art) es un entorno de programación gráfico basado en el lenguaje Logo (http://en.wikipedia.org/wiki/Logo_(programming_language)), en el que se pueden hacer pequeños programas y realizar diseños con una tortuga de forma gráfica.

Desde la versión 106, TurtleArt soporta plugins que permiten mejorar y agregar nuevas “primitivas” a nuestro software. En este artículo explicaremos la creación de un plugin personalizado para TurtleArt. Este artículo requiere cierto conocimiento previo en programación orientada a objetos, Python y la forma de trabajo de Sugar (http://en.wikipedia.org/wiki/Sugar_(desktop_environment)).

## Preparando el entorno

Primero debemos tener instalado en nuestra computadora un entorno Sugar (no es excluyente pero siempre es conveniente);

En la distribucion GNU/Linux Fedora es tan simple como poner:

```
$ su -c ‘yum install sugar*’
```

o también:

```
su -c ‘yum groupinstall sugar’
```

Con esos comandos nuestro sistema Linux instalará todo el entorno Sugar con sus actividades y el emulador **sugar-emulator** (muy cómodo para las pruebas).

A continuación descargaremos la última versión de TurtleArt desde la página web:

*http://activities.sugarlabs.org/en-US/sugar/downloads/latest/4027/addon-4027-latest.xo?src=addo*

También pueden descargar del siguiente link la versión de TurtleArt con el plugin **“tortucaro”** (Icaro + TurtleArt) para estudiarlo tranquilos (versión 106 al momento de la edición de esta revista).

*http://valentinbasel.fedorapeople.org/TurtleArt.tar*

***Atención: esta versión no es igual a la que analizamos en este artículo.***

Luego de realizar la descarga, desempaquetamos el archivo **.tar** en nuestro **home** y procedemos a ver la estructura de directorios.

Los archivos que nos importa “tocar” están dentro del directorio **plugins**.

La idea de esta versión de TurtleArt es posibilitar el desarrollo de nuevas paletas por parte de los desarrolladores sin necesidad de modificar el código fuente original del software (como en las versiones anteriores). Así, antes de la versión 107, para hacer nuestro plugin básicamente teníamos que crear una archivo con el siguiente formato:

```
<NOMBRE>_plugin.py
```

Donde **“<NOMBRE>”** es la denominación que le daremos a nuestro plugin, por ejemplo: “prueba_plugin.py”.

A partir de la versión 107, se mejoró el sistema para acomodar todos los archivos del plugin directamente en una carpeta dentro del directorio plugins.

## Desarrollando un plugin

Vamos a crear una carpeta de nombre **“prueba”** y dentro crearemos un archivo **“prueba/prueba.py”**. En nuestro archivo implementaremos una clase de Python con el mismo nombre que el archivo, y solamente pondremos la primera letra en mayúscula: “Prueba” (**hay que respetar ese formato**).

Ahora iremos agregando el siguiente código al archivo

```
# es necesario importar todos estos módulos para poder trabajar con
# TurtleArt

import gst
import gtk
from fcntl import ioctl
import os
from gettext import gettext as _
from plugins.plugin import Plugin
from TurtleArt.tapalette import make_palette
from TurtleArt.talogo import media_blocks_dictionary, primitive_dictionary
from TurtleArt.tautils import get_path, debug_output
import logging

_logger = logging.getLogger('TurtleArt-activity prueba plugin')


class Prueba(Plugin):

    def __init__(self,parent):
        self._parent = parent
        self._status = False
```

La primera parte es fundamental para poder importar todos los módulos internos de TurtleArt; aquí, por razones de extensión, no explicaré en detalle de que se trata cada módulo.

Con nuestra clase creada e inicializada (`def__init__`), procedemos a colocar el método `setup`

```
# dentro ce la clase Prueba
def setup(self):
    palette = make_palette('prueba',
                           colors=["#006060", "#A00000"],
                           help_string=_('esta es una prueba'))
```

Acá declaramos una nueva paleta de nombre `prueba`; el campo `colors` representa los dos colores con los que se hará el gradiente del botón que muestra TurtleArt (para diferenciarlos de las otras “primitivas” del sistema). El campo `help_string` es bastante entendible, básicamente se trata de información sobre la paleta (pero no sobre los botones de esa paleta, como ya veremos).

```
# dentro ce la clase Prueba y dentro del metodo setup

#[1]
primitive_dictionary['boton'] = self._boton_prueba

#[2]
palette.add_block('boton',
            style='basic-style-1arg',
            label=_('boton'),
            prim_name='boton',
            help_string=_('boton de prueba'))

#[3]
self._parent.lc.def_prim('boton', 1, lambda self, valor: primitive_dictionary['boton'](valor))
```

Este es nuestro primer botón, acá definimos su comportamiento, estilo, nombre y función a la que apunta.

en **[1]** definimos una “primitiva” para el diccionario. Cuando la coloquemos, el sistema llamará a `_boton_prueba` (la función donde estará el código de acción).

Para **[2]** `Palette.add_block` crea nuestro botón, el cual estará dentro de nuestra paleta `prueba`; el primer campo es el nombre del botón. Con `style` definimos que tipo de botón sera, si tendrá o no argumentos, si en lugar de enviar información la recibirá o si es un estilo especial. Para saber que tipos de estilos se pueden usar hay que revisar el archivo **“tapalette.py”** dentro del sub directorio **“TurtleArt”**, en nuestro directorio principal en el que descomprimimos el **.tar** (Son bastantes, nosotros sólo usaremos el estilo básico con un argumento `basic-style-1arg`). El campo `label` es el nombre que mostrara el botón en su cuerpo; en el campo `prim_name` escribo lo mismo que en el primer campo (`boton`); `help_string` es el texto de ayuda que mostrará TurtleArt cuando pasemos el mouse por arriba del botón. la línea **[3]** de código es donde el sistema enlaza el botón con la función que definamos. Lamdba es una interesante sintaxis para definir funciones mínimas implementada por python, lo que hacemos es pasar a la función `_boton_prueba` mediante `primitive_dictionary['boton']` la variable `valor`. Como es un botón básico

de un argumento, permite poner una caja de valor y almacenarlo en dicha variable.

Finalmente, lo único que nos falta es la función `_boton_prueba` donde haremos toda la lógica del botón propiamente dicho.

```
# dentro ce la clase Prueba
def _boton_prueba(self,valor):
   print "el valor del boton ", valor
```

La función es muy sencilla, lo único que hace es mostrar un poco de código trivial para ilustrar el funcionamiento de un plugin.

Como verán, programar plugins para TurtleArt no es de lo más complejo. Si bien hay poca documentación y la mayoría está en inglés, con un poco de paciencia se pueden armar un montón de paletas personalizadas para las más variadas actividades.

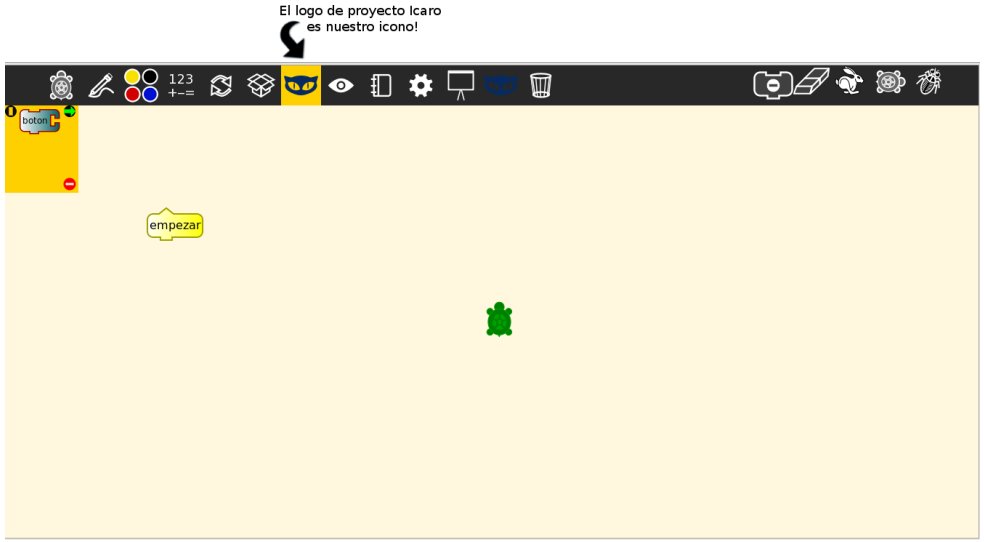
Con nuestro archivo **"prueba.py"** ya terminado lo único que nos falta es crear la carpeta **"prueba/icons"**, quedando el árbol de directorio de la siguiente manera:

```
~/TurtleArt/
    plugins/
        prueba/
            icons/
```

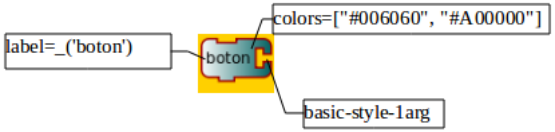
Dentro de **icons*** hay que colocar dos archivos **.svg** de 55x55 píxeles con el nombre **pruebaon.svg** y **pruebaoff.svg** (el mismo nombre del plugin).

Con eso debería aparecer el icono **pruebaoff.svg** dentro de la barra de herramientas de TurtleArt.

**¡TurtleArt con nuestro botón de prueba andando!**




**Detalle del armado del botón dentro de la paleta.**




Para terminar les presento el archivo **prueba.py** completo

```
import gst
import gtk
from fcntl import ioctl
import os
from gettext import gettext as _
from plugins.plugin import Plugin
from TurtleArt.tapalette import make_palette
from TurtleArt.talogo import media_blocks_dictionary, primitive_dictionary
from TurtleArt.tautils import get_path, debug_output

import logging
_logger = logging.getLogger('TurtleArt-activity prueba plugin')

class Prueba(Plugin):
    def __init__(self, parent):
```

```
        self._parent = parent
        self._status = False

    def setup(self):
        palette = make_palette('prueba',
                               colors=["#006060", "#A00000"],
                               help_string=_('esta es una prueba'))
        primitive_dictionary['boton'] = self._boton_prueba
        palette.add_block('boton',
                    style='basic-style-1arg',
                    label=_('boton'),
                    prim_name='boton',
                    help_string=_('boton de prueba'))
        self._parent.lc.def_prim('boton', 1,
                                 lambda self, valor: primitive_dictionary['boton'](valor))

    def _boton_prueba(self,valor):
       print "el valor del boton ", valor
```

# Chorizos de serpiente


**Autor:** Juanjo Conti

**Bio:** Juanjo es Ingeniero en Sistemas. Programa en Python desde hace 5 años y lo utiliza para trabajar, investigar y divertirse.

**Web:** http://juanjoconti.com.ar

**Email:** jjconti@gmail.com

**Twitter:** @jjconti

Luego del evento, aunque estábamos todos bastante cansados es costumbre de este grupo hacer algún encuentro social en la noche de los días de evento. En esta oportunidad, los organizadores iban a agasajarnos con una choripaneada.

Apenas se cerraron las salas los locales se fueron a bañar y los visitantes caminamos ciudad arriba en busca un un bar. Ya en Nueva Córdoba, luego de evaluar un par de opciones terminamos en un lugar con nombre italiano en el que algunos tomaron gaseosa, otros cerveza y otro café :) Alguno partían pronto y no podían quedarse a la choripaneada, así que se comieron unas pizzas.

La cena fue en un salón de la facultad, el mismo en el que habíamos almorzado. Llegamos temprano y no había llegado mucha gente aún. Por suerte había un metegol que hizo nuestra delicia (hay un video en youtube: http://www.youtube.com/watch?v=nRwlo6WbiQM).

Pronto empezó a llegar más gente con la que hubo mucha charla y un poco mas tade los chorizos empezaron a salir de parrilla. En el medio del deleite parrillero empezamos otra competencia deportiva: Ping pong.

Muy buena noche, muy buena cena y charlas muy interesantes.

Esperamos la próxima!

# Procesamiento de lenguaje natural con NLTK (Natural Language Toolkit)


**Autor:** Rafael Carrascosa

**Bio:** Rafa se recibió el año pasado de licenciado y actualmente es estudiante de doctorado en el grupo de PLN de FaMAF, donde es profe en Redes. Le gusta programar y meterle mano a las cosas.

**Email:** rafacarrascosa@gmail.com


**Autor:** Pablo Duboue

**Bio:** Pablo se recibió en FaMAF en 1998. Se siguió recibiendo en la Universidad de Columbia en Nueva York en 2005 (doctorado en generación de Lenguaje Natural). De ahí estuvo en IBM Research del 2005-2010, donde trabajó en Preguntas y Respuestas, búsqueda de Expertos y DeepQA (Jeopardy!). Por estos días vive en Montreal, Canadá, donce es socio de un Hackerspace (Foulab) y está empezando algunos micro-emprendimientos.

**Email:** pablo.duboue@gmail.com

**Twitter:** @pabloduboue

**Webpage:** http://www.duboue.net/pablo/

**¿Qué es el Procesamiento de Lenguaje Natural?** El PLN (o NLP en inglés) es el uso de computadoras para el procesado de lenguaje humano. Tiene pilas de aplicaciones prácticas (blogs, twitter, phones, etc) y en algún sentido puede considerarse un hito (casi) final para la Inteligencia Artificial (El test de Turing, al fin y al cabo, es lenguaje natural, pero esto es muy discutido, un robot immerso en el mundo es mucho más difícil).

Todo muy lindo con el PLN, pero la gente es muy buena hablando, por naturaleza (su lengua materna, por supuesto), así que los usuarios de PLN tienen altas espectativas del rendimiento de los sistemas (que no se conllevan con la realidad). Aparte, las personas que hacen PLN tienen una mezcla de interéses tanto en lenguajes como en matemáticas, una combinación inusual. Hoy en día el PLN involucra pilas de ingeniería y tweaking (y la sensación de que... ¡Tiene que haber una forma mejor!).

## Natural Language Toolkit

**NLTK** es un toolkit, una colección de paquetes y objetos python muy adaptados para tareas de PLN. El NLTK es a la vez una herramienta que introduce a nuevas personas al estado del arte en PLN mientras que permite a los expertos sentirse muy cómodos dentro de su entorno. Comparado a otros frameworks, el NLTK tiene asunciones por defecto muy fuertes (por ej, un texto es una secuencia de palabras), aunque pueden cambiarse. NLTK no sólo se centra en gente que trabaja en PLN viniendo desde la computación sino también en lingüistas haciendo trabajo de campo. NLTK se mezcla bien con Python (no está simplemente "implementado" en Python).

**Algunos datos acerca del NLTK:**

> *El NLTK comenzó en la Universidad de Pennsylvania, como parte de un curso de lingüística computacional. Se encuentra disponible en Internet, desde http://www.nltk.org/. Su código fuente se distribuye bajo la Apache Licens version 2.0 y hay un libro de 500 paginas por Bird, Klein y Loper disponible desde O'Reilly "Natural Language Processing with Python", muy recomendable (http://nltk.googlecode.com/svn/trunk/doc/book/).*
>
> *El toolkit incluye también datos en la forma de colecciones de textos (muchos anotados) y modelos estadísticos.*

## NLTK y Python

El toolkit se integra muy bien con Python ya que trata de hacer la mayoria de las cosas con facilidades de Python como porciones y listas por comprensión. Por ejemplo, un texto es una lista de palabras y una lista de palabras puede ser transformada en un texto. Los objetivos de diseño del toolkit (Simpleza, Consistencia, Extensibilidad y Modularidad) van de la mano con el diseño mismo de Python. Además, fiel a dichos objetivos, NLTK evita crear sus propias clases cuando los diccionarios, listas y tuplas por defecto de Python son suficientes.

**Paquetes principales de NLTK:**

- **Acceso a documentos:** Interfaces para colecciones de textos.
- **Procesamiento de cadenas:** Tokenizado, detección de oraciones, stemmers.

- **Descubrimiento de collocactions:** Tokens que aparecen juntos más frequentemente que por chance.
- **Part-of-speech tagging:** Distinguir substantivos de verbos, etc.
- **Clasificación:** Clasificadores general, basados en diccionarios de Python como entrenamiento.
- **Chunking:** Partir una oración en unidades granulares
- **Análisis Sintáctico:** Análisis complejo (sintáctico y otros).
- **Interpretación Semántica:** Cálculo λ (lambda), lógica de 1er órden, etc.
- **Métricas de Evaluación:** Precisión, covertura, etc.
- **Estadística:** Distribución de frecuencias, estimadores, etc.
- **Aplicaciones:** Browser de WordNet, chatbots.

## Algunos Ejemplos

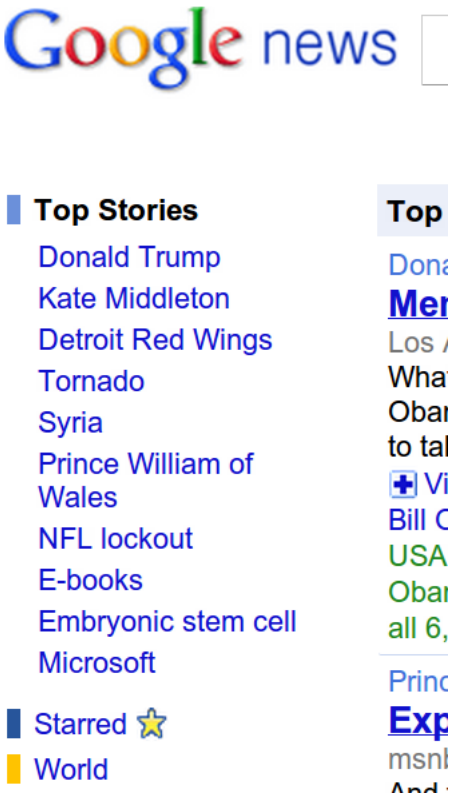
Para probarlo, el importar todo lo que está en el paquete nltk.book pone al intérprete de Python listo para empezar. (También hay que descargar primero los datos binarios, importando `nltk` y haciendo `nltk.download()`).

Una vez hecho esto, se puede preguntar, por ejemplo, por palabras similares basadas en contextos, dado un texto.

```
>>> from nltk.book import *
# long comment, skipped
>>> moby_dick = text1
>>> moby_dick.similar('poor')
Building word-context index...
old sweet as eager that this all own help peculiar german crazy three
at goodness world wonderful floating ring simple
>>> inaugural_addresses = text4
>>> inaugural_addresses.similar('poor')
Building word-context index...
free south duties world people all partial welfare battle settlement
integrity children issues idealism tariff concerned young recurrence
charge those
```

## Tutorial: Noticias destacadas

Nos interesa hacer algo similar a las Top Stories de http://news.google.com/.




**Top stories a la NLTK**

**Nuestra estrategia será:**

- Buscaremos Named Entities
- Usando NLTK out-of-the-box
- Luego puntuaremos las entidades y mostraremos las mejores.

Como datos usamos **7,705** noticias internacionales prestadas del sitio de Reuters U.S. (http://www.reuters.com/)

**¿Qué son las entidades nombradas?**

Las entidades nombradas son elementos atómicos del texto que caen en categorias tales como personas, lugares, organizaciones. Para extraer entidades nombradas con nltk usamos el siguiente código:

```
>>> import nltk
>>> s = """
... Prince William and his new wife Catherine kissed twice
... to satisfy the besotted Buckingham Palace crowds"""
>>> a = nltk.word_tokenize(s)
>>> b = nltk.pos_tag(a)
>>> c = nltk.ne_chunk(b,binary=True)
>>> for x in c.subtrees():
...     if x.node == "NE":
...             words = [w[0] for w in x.leaves()]
...             name = " ".join(words)
...             print name
...
Prince William
Catherine
Buckingham Palace
>>>
```

Los algoritmos del estado del arte para el reconocimiento de entidades nombradas automáticamente se adaptan entidades que no han sido vistas con anterioridad. De la imagen de **news.google.com** que presentamos antes vemos que la lista está comprendida en su mayor parte por entidades nombradas. Entonces encontrar entidades nombradas y ponerlas en un ranking nos permitiría simular el comportamiento de **news.google.com**.

El código anterior en tres simples pasos:

1. Armar una lista de palabras

```
>>> a = nltk.word_tokenize(s)
['Prince', 'William', 'and', 'his', 'new', 'wife', ...
```

1. Agregar la categoría gramatical

```
>>> b = nltk.pos_tag(a)
[('Prince', 'NN'), ('William', 'NNP'), ('and', 'CC'), ...
```

1. Anotar las entidades nombradas

```
>>> c = nltk.ne_chunk(b,binary=True)
Tree('S',[Tree('NE', [('Prince', 'NN'), ('William', 'NNP')]), ...
```

El resto del código del ejemplo es solo para mostrar las entidades en pantalla.

**¿Cuáles más son relevantes?**

El problema pendiente es construir un ranking con las entidades más relevantes.

¿Como distingir que **"Japon"** es más relevante entre el 10 y 20 de marzo (recordar el tsunami) que entre el 10 y 20 de abril? Para responder esto, el método que proponemos para este ejemplo consiste en considerar cuando más frecuente es Japon en un rango de días con respecto a lo que habitualmente es. Entonces, nuestra función de ranking es:

$$ratio(word) = \frac{prob.\ de\ word\ en\ los\ días\ i..j}{prob.\ de\ word\ en\ todas\ las\ noticias}$$

Las palabras con ratio más alto fueron significativas en los dias entre `i` y `j`.

**¿Cómo evitar superposición?**

**"Japon"** y **"Tokio"** podrían tener ratio alto pero probablemente solo nos interese nombrar solo una como noticia relevante.

Para resolver esto hacemos:

- Elegimos la entidad **E** con mayor ratio.
- Tiramos todos las noticas en las que **E** aparece.
- Recalculamos los **ratios** y volvemos al primer paso.

**Un ouput típico de todos los pasos es**:

```
============================================================
Top news between 2011-03-11 00:00:00 and 2011-03-20 00:00:00
============================================================
        zuwarah
        uss ronald reagan
        richard wakeford
        patrick fuller
        unit
        soviet ukraine
        g7
        tokyo commodity exchange
```

```
        roppongi
        nuclear security
```

El código del ejemplo esta disponible en:

*http://duboue.net/download/pyday-nltk.tar.gz*

## Conclusiones

**Rafael Carrascosa** (A.K.A. Player 1) usó NLTK en su trabajo en la Facultad de Matemática, Astronomía y Física de la UNC, principalmente en un pipeline sobre avisos clasificados para:

- Para hacer POS tagging del castellano.
- Para hacer Chunking del castellano.
- Para hacer chunk sense disambiguation.

Además, tambien armó modelos de lenguaje para compresión de textos usando gramáticas probabilisticas, N-Gramas y parsers estadísticos, todos basados en NLTK.

Entonces... **¿Cuanto se la banca NLTK?** Al estar implementado en puro Python, hay veces que se toma su tiempo, aparte los distintos paquetes disponibles varian mucho a nivel de madurez del código. Una pregunta que se puede hacer es uno es:

> ***¿Se puede usar para implementar productos comerciales?***
>
> *A nivel de licencia, está todo bien. A nivel de errores de los anotadores, hay que probar y a nivel de velocidad, también hay que probar. O sea... **¡Probá!***

Otros frameworks que andan dando vuelta incluyen **GATE** y **UIMA** que son muy basados en **Java**.

**¿Cómo continuar?** NLTK está muy bueno, probálo y fijate el libro (que está en linea).

# Haciendo videojuegos con pilas


**Autor:** Hugo Ruscitti

**Bio:** Estudiante de Informática en la Universidad Tecnológica Nacional (UTN). Ha trabajado elaborando distribuciones de Linux y actualmente es socio de la cooperativa de trabajo gcoop. Su mayor hobby es la programación de videojuegos, actividad que difunde y realiza junto a los miembros de su sitio http://www.losersjuegos.com.ar/

**Email:** hugoruscitti@gmail.com

**Web:** www.pilas-engine.com.ar


En este artículo quiero contarles brevemente algo sobre el desarrollo de videojuegos, por qué me parece una idea interesante para llevar a las escuelas y cómo se pueden poner en práctica haciendo videojuegos fácilmente.

## ¿Por qué hacer videojuegos?

Desarrollar videojuegos es una actividad super interesante, consiste en llevar adelante un proyecto, darle vida a una idea, contar una historia y poner en práctica un montón de conceptos útiles sobre computadoras, matemáticas, literatura, física, etc.

Y también están dirigidos a un público, porque los videojuegos en sí, son muy llamativos, a todas las personas que le cuento sobre videojuegos le encanta la idea, y se suelen entusiasmar mucho compartiendo cómo se divierten o divirtieron con tal o cual videojuego.

Por ese motivo, creo que podemos comenzar a ver a los videojuegos como algo mas completo que solamente diversión. Porque también es una oportunidad muy valiosa para aprender y despertar el interés de los chicos en las aulas.

## Es mas fácil de lo que parece...

Pero casi siempre que hablamos de programación, muchas personas se tienen que enfrentar a muchos miedos y pre-conceptos algo desacertados. Hace algún tiempo realizar programas era una tarea muy compleja, pero hoy no es así.

Por suerte Python, junto a bibliotecas de alto nivel y el acceso a la tecnología están dando a las personas cada vez mas herramientas para hacer cosas creativas y novedosas con sus computadoras.

## Pilas

**Pilas** es una biblioteca desarrollada para facilitar el desarrollo de videojuegos a todas las edades.

Está orientado principalmente a los mas jóvenes, que recién descubren las computadoras y quieren hacer algo creativo con ellas.

## Creando una ventana y muchos actores

Veamos un ejemplo desde la consola interactiva de python: `pilas` se utiliza como cualquier módulo que podemos incorporar y comenzar a utilizar así:

```
import pilas
pilas.iniciar()
```

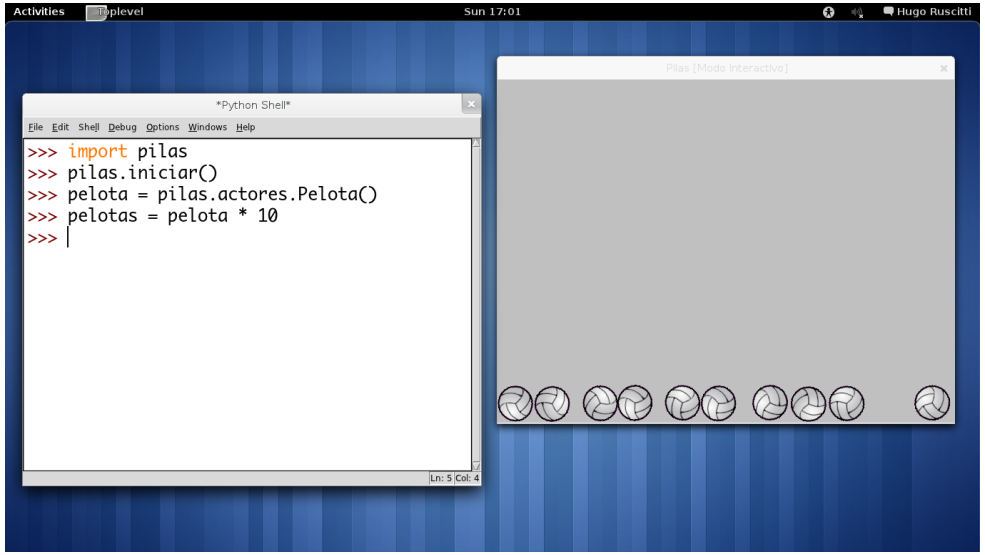
La función `iniciar` se encarga de abrir una ventana, y nos ofrece algunos argumentos opcionales por si queremos especificarlos (este no es el caso.)

Una vez que tenemos la ventana, podríamos crear algún personaje animado para tener algo de acción:

```
pelota = pilas.actores.Pelota()
```

Lo que veremos en la pantalla es una pelota de volley rebotando por si sola, sobre el suelo. Y si queremos que rebote con alguna otra cosa podríamos multiplicarla, para tener un montón de pelotas rebotando:

```
pelotas = pelota * 10
```




¿Y que hay de la gravedad?, se puede cambiar fácilmente así:

```
pilas.fisica.definir_gravedad(10, 30)
```

Dónde 10 y 30 son la aceleración horizontal y vertical respectivamente. Por cierto, los valores habituales suelen ser 0 y -90.

## Un ejemplo mas específico

Veamos esto un poco mas en detalle, escribe `pilas.reiniciar()` para limpiar lo que tenemos en pantalla.

Ahora bien, los actores son objetos que viven en el módulo `pilas.actores`. Por ejemplo si escribimos:

```
mono = pilas.actores.Mono()
```

Aparecerá un monito en el centro de la ventana, y cómo lo creamos usando una referencia vamos a poder indicarle cosas cómo las siguientes:

```
mono.gritar()
mono.decir("Aguante la revista PET!!!")
```


También podríamos alterarle algunas propiedades visuales cómo su rotación, tamaño, posición, transparencia etc. Tomemos este ejemplo, vamos a mover al mono hacia la derecha de la pantalla y dupliquemos su tamaño:

```
mono.x = 100
mono.escala = 2
```

¿Notaste que los cambios son inmediatos?, ¿Cómo hacemos para crear animaciones?. Simplemente tienes que cambiar los número enteros por listas:

```
mono.x = [0]
mono.y = [200, 0]
mono.rotacion = [360]
```

La primer sentencia mueve al mono al centro de la ventana (horizontalmente), la segunda sentencia hace que el mono se mueva hacia arriba y luego hacia abajo. Y la última sentencia hace girar al mono una vuelta completa.

## Todos los actores son objetos

El ejemplo anterior nos muestra que los actores, en realidad, son objetos: Tienen propiedades cómo `escala`, `x` e `y`, pero también tienen comportamiento, cómo los métodos `decir` o incluso la interpretación de mensajes cómo * (igual que los números y los strings) cómo vimos en el ejemplo de la pelota de volley.

Esta es una idea muy poderosa de la programación, porque significa que cuando logras dominar un actor, en realidad, estás aprendiendo a manejar muchos actores, y a su vez, estás programando python!

## Investigando

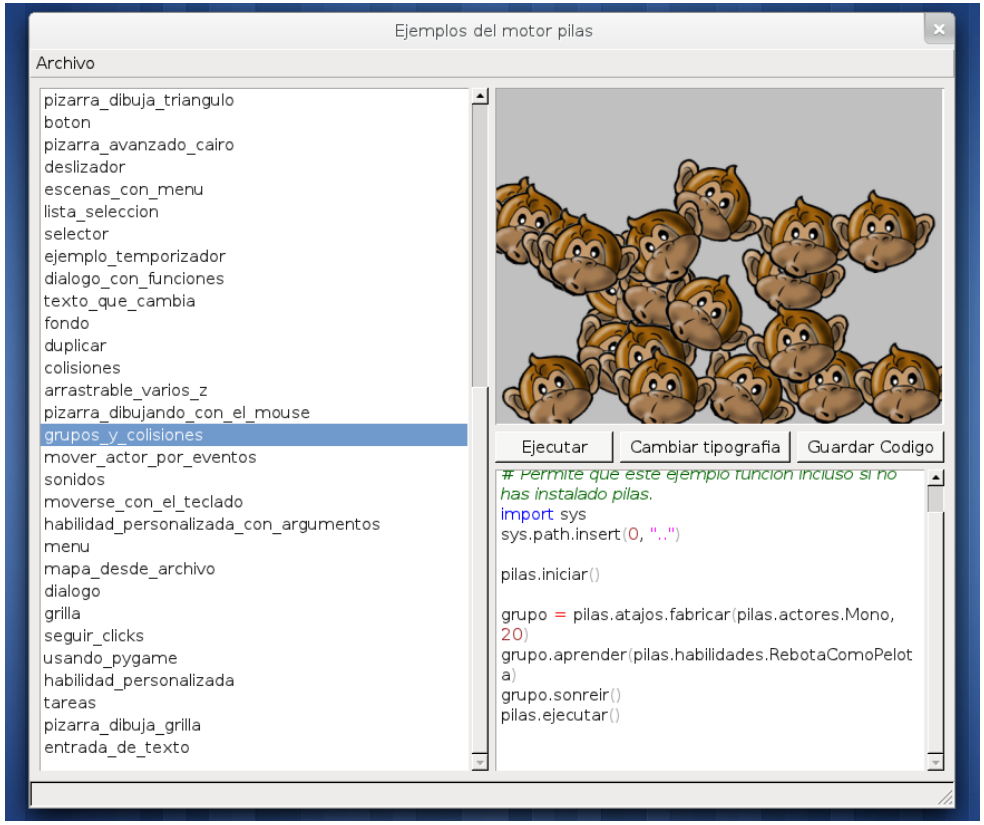
Hay una función de `pilas` que viene útil para comenzar a investigar: la función `pilas.ver`:

```
pilas.ver(mono)
```

Esta función se puede usar para inspeccionar el código fuente de cualquier cosa: módulos, funciones, actores, referencias a objetos a clases etc... ¿cómo funciona?, fácil, hace `pilas.ver(pilas.ver)`.

Y por supuesto también está la función `help` y el auto-completado de código.

Incluso, si escribes `pilas -e` en una terminal, observarás una aplicación que te permite ver todos los ejemplos de código que incluye pilas:




## Mirando hacia el futuro

Creo que estamos ante una posibilidad interesante de mostrar lo divertido e interesante que resulta la programación. Una posibilidad de entregar a las personas una herramienta muy poderosa para pasar de consumidores a productores de tecnología.

Sólo que en este punto, no se trata de algo estrictamente técnico, porque el desafió está precisamente en difundir, crear y ayudar dentro de `pyar` para que las personas se animen a participar.

Por ese motivo, si te gustó lo que viste en este artículo, mi consejo es que te animes a escribir y contarle a mas personas sobre el desarrollo de juegos.

Podrías visitar la web de pilas (www.pilas-engine.com.ar), contarnos por la lista de correo de pyar que te pareció, o cualquier idea que tengas. Tus comentarios son muy valiosos para que este y otros proyectos puedan seguir adelante.

# pycamp ORM, haciendo *fácil* usar sqlalchemy


**Autor:** Emiliano Dalla Verde Marcozzi

**Bio:** El autor se inicio en Python de la mano de Plone, siendo a su vez sus primeros pasos en la programación.

**Email:** edvm@airtrack.com.ar

**Twitter:** @edvm

**Webpage:** http://ninjasandpythones.blogspot.com/

## ¿De que se trata este articulo?

Esencialmente de tres cosas:

- ORM / Object Relation Mapper, ¿que es?, ¿para que sirve?
- SQLAlchemy, un ORM en Python
- pycamp.orm, 140 lineas (aproximadamente) que *facilitan* usar SQLAlchemy

¿Para que son estas tres cosas? Para trabajar con bases de datos relacionales, de diferente fabricantes como son:

- PostgreSQL
- MySQL
- Oracle
- SQLite

## Nuestro propósito

¿Conquistar el mundo? Si, podría ser, pero no lo es en este artículo ;). Nuestro propósito es poder trabajar con bases de datos relacionales utilizando Python y para tener un panorama general de los diferentes actores que intervienen en esta escena, pues debemos conocer a:

- Jenna Jameson, erghh digo, **SQL / Structured Query Language**
- **Base de Datos** (PostgreSQL, MySQL, etc)
- **Conector / Driver** de Python para conectarnos contra la Base de Datos
- **Python**

El lenguaje **SQL / Structured Query Language** es un estándar ANSI (American National Standars Institute) el cual nos permite consultar, definir y modificar datos en una DB relacional. Ahora, esto esta *buenísimo!*, tenemos un estándar!!! Osea: las bases de datos que implementen este estándar podrán entender los comandos que programemos en este lenguaje.

Segundo: **Base de Datos** que implementan en **PARTE** el SQL. Esto quiere decir que pueden tener otras *palabras* agregadas al lenguaje SQL que implementan, por ejemplo MySQL puede tener sentencias que no existen en PostgreSQL.

**Conector / Driver** (ejemplos son psycopg2, MySQLdb) es código que permite que Python se comunique con la Base de Datos. Las tareas que contempla un conector es manejar por ejemplo la parte de sockets, y otras muchas operaciones de las cuales no tengo ni idea ;).

**Python** lenguaje de programación todopoderoso con el cual podríamos conquistar el mundo, pero en este articulo lo utilizaremos para trabajar con bases de datos relacionales.

Conocidos los actores, podríamos crear un súper diagrama en ASCII que represente como se relacionan estos actores entre si:

```
MySQL <------------> sql_lang_1 <---> MySQLdb <----> Python
PostgreSQL <-------> sql_lang_2 <---> psycopg2 <---> python
Oracle <-----------> sql_lang_3 <---> cx_Oracle <--> Python
```

Explicando el primer caso de nuestro súper diagrama en ASCII, y leyéndolo desde derecha a izquierda: contamos con nuestro todopoderoso lenguaje *Python*, cual mediante el conector *MySQLdb*, realiza consultas utilizando *sql_lang_1* a la base de datos *MySQL*. ¿Simple no?

## Mismas consultas, diferentes bases de datos

Vamos a plantear una hipotética situación en la cual debemos crear un programa el cual precisa guardar los datos que genera en una base de datos relacional.

Resulta además que la empresa donde trabajamos esta utilizando MySQL, con lo cual nos disponemos a escribir nuestras consultas utilizando el conector MySQLdb.

Luego de 1337 meses de desarrollo tenemos nuestro programa completamente funcional, pero resulta que nuestro/a jefe/a se ha levantado con un humor para nada compatible con MySQL y da la directiva de que debemos utilizar PostgreSQL y que todo software que utilice una base de datos distinta de PostgreSQL sera condenado a jugar en la Arena

de TRON!

Ok, nosotros y nuestro programa se encuentran en una situación complicada, hemos de cambiar todas las consultas que hemos realizado para MySQL para que funciones con PostgreSQL o conseguirn rápidamente un *frisbees* (http://en.wikipedia.org/wiki/Flying_disc) y comenzar a practicar para cuando nos toque combatir en la Arena de TRON (http://en.wikipedia.org/wiki/Tron_(film)).Al margen de la quizá cómica situación y para ajustarme un poco a la charla que se dio en el PyDay de Córdoba, podríamos decir que:

- Las bases de datos tienen cada una su propio dialecto.
- Con Python, utilizamos drivers/conectores para poder hablar el dialecto de la DB a la cual nos conectamos y utilizamos.
- El escribir en nuestro código SQL utilizando el dialecto de la DB parece no ser una buena idea, puesto que al cambiar de motor / engine de DB nos vemos obligados a tener que pasar todas nuestras queries al dialecto de la nueva DB.

## SQLAlchemy, el ORM utilizado por MacGyver y Mario Baracus

A toda la problemática planteada anteriormente de que veníamos escribiendo queries según el engine/dialecto de nuestra DB, existe una solución y esta se llama **SQLAlchemy**.

SQLAlchemy es un ORM y un ORM es un *O*bject *R*elation *M*apper, osea: un *mapeador/conversor* de las relaciones de los datos de nuestra DB a objetos. Por sino quedo claro, veamos un ejemplo:

Supóngase una tabla “Persona” en nuestra DB cual tiene las columnas:

- Nombre de tipo `VARCHAR`.
- FechaNacimiento de tipo `DATE`.

Hasta aquí es la parte relacional, ahora si nosotros representáramos esta estructura de datos con objetos en python, bien podríamos tener lo siguiente:

```
import datetime

class Persona(object)
    """ I'm not only a cute face, i'm a person!
    """
    nombre = ''
    borndate = datetime.datetime.now()
```

El trabajo del ORM es justamente *leer* las tablas, columnas, propiedades y las relaciones entre datos para expresarlos utilizando objetos Python. Esto para nosotros es muy bueno, puesto que como veremos mas adelante, vamos a trabajar con objetos Python y no vamos a estar escribiendo SQL. Será entonces responsabilidad del ORM traducit todo a dicho lenguaje para interactuar con la DB. Veamos esto con un súper gráfico ASCII iluminador del entendimiento:

```
DATABASE <-----> SQLALCHEMY(ORM) <------> PYTHON
```

Entonces: Escribimos python y SQLAlchemy *traduce* lo traduce al dialecto que utiliza nuestra DB. Por ejemplo: Un *select* en nuestra tabla Persona se escribe:

```
>>> from edo import Session
>>> from limbo import Persona

>>> session = Session()
>>> first_person = session.query(Persona).first()
>>> first_person.name
'Lady Gaga'
```

En el ejemplo hemos importado nuestra clase Persona y un ficticio Session con el cual hemos realizado nuestra consulta. Note-se que se trabajó con Python y no con el dialecto de nuestra DB (de ello se encarga SQLAlchemy ;)).

Con esto queremos decir que el código python escrito anteriormente, gracias a SQLAlchemy, funciona en PostgreSQL, MySQL o en Oracle sin tener que modificar nada, ¿No es bonito eso ? ^^’

## Haciendo Alquimia, ingredientes

Para comenzar a utilizar SQLAlchemy es preciso entender los siguientes cuatro elementos: Los cuatro elementos clásicos griegos (tierra, agua, fuego y aire) datan de los tiempos presocráticos y perduraron a través de la Edad Media hasta el Renacimiento influenciando profundamente la cultura y el pensamiento europeo, pero no así en SQLAlchemy; puesto que los cuatro elementos fundamentales a entender son:

- Sesiones

- Mapper
- MetaData
- Engine

**Sesiones**: Las sesiones están hechas para *abrir y cerrar* conexiones contra la DB. Esto en si no es verdad, puesto que SQLAlchemy mantiene un **pool** de conexiones persistentes contra la DB con lo cual cuando creamos una sesión, estaremos tomando un elemento de este pool de conexiones y al cerrar nuestra sesión, estaremos retornando la conexión al pool de conexiones. Por lo dicho anteriormente, crear sesiones es muy rápido y barato a nivel recursos, porque las conexiones ya están previamente hechas y disponibles en nuestro pool de conexiones.

**Mapper**: Es un código de tecnología de punta, encargado justamente de *mapear* las estructuras y propiedades de nuestra DB a objetos Python y viceversa.

**MetaData**: Por ahora, lo dejaremos como un objeto *oscuro*.

**Engine**: Es un objeto en el cual seteamos las propiedades de la conexión contra nuestra base de datos, como ser: el usuario, password, nombre de la base de datos y motor de la base de datos (Oracle, MySQL, PostgreSQL, etc). También podemos setear la cantidad de conexiones persistentes a tener en nuestro pool de conexiones y muchas chucherías mas.

## Orden de los ingredientes

Como en todo procedimiento *alquimístico*, el orden es importante y es por ello que a continuación detallamos cual es el orden de los ingredientes de SQLAlchemy:

1. Crear el engine
2. Bindear contra nuestro engine
3. Listo :)

Lo primero que hacemos es declarar el engine. Y pues claro! Es ahí donde definimos el usuario, password, host al que conectarnos, tipo de DB y todas esas cosas; así que tiene algo de sentido que esto sea lo primero a realizar.

```
[1] from sqlalchemy import create_engine

[2] url = 'mysql://user:passwd@host/pet'
[3] engine = create_engine(url)
```

En la linea **[1]** importamos create_engine desde sqlalchemy, en la linea **[2]** creamos una *url* cual define los datos de conexión siendo los mismos:

```
url = 'TIPODEDATABASE://USUARIO:PASSWD@IPHOST/DATABASENAME'
```

Y, por último en el paso **[3]** llamamos al método `create_engine` con nuestra url como parámetro

Ahora veamos esto con un ejemplo de código real.

Para ello voy a estar utilizando un *virtualenv* y dentro del mismo instalaré sqlalchemy (para saber mas acerca de crear virtualenvs, ver: http://pypi.python.org/pypi/virtualenvwrapper)

```
edvm@Yui:~$ mkvirtualenv --no-site-packages pet
New python executable in pet/bin/python
Installing  distribute.....................................................done.
virtualenvwrapper.user_scripts creating /home/edvm/.venvs/pet/bin/  predeactivate
virtualenvwrapper.user_scripts creating /home/edvm/.venvs/pet/bin/postdeactivate
virtualenvwrapper.user_scripts creating /home/edvm/.venvs/pet/bin/preactivate
virtualenvwrapper.user_scripts creating /home/edvm/.venvs/pet/bin/postactivate
virtualenvwrapper.user_scripts creating /home/edvm/.venvs/pet/bin/get_env_details
(pet)edvm@Yui:~$ pip install ipython
Downloading/unpacking ipython
...
...
Successfully installed ipython
Cleaning up...
(pet)edvm@Yui:~$ pip install sqlalchemy
Downloading/unpacking sqlalchemy
....
....
Successfully installed sqlalchemy
Cleaning up...
```

Ahora ejecuto ipython y comenzamos a trabajar:

```
(pet)edvm@Yui:~$ ipython
....

In [1]: import os
In [2]: from sqlalchemy import create_engine
```

```
In [3]: db = os.path.join(os.path.abspath(os.path.curdir), 'db.sql')

In [4]: engine = create_engine('sqlite:///%s' % db)
In [5]: engine
Out[5]: Engine(sqlite:////home/edvm/db.sql)
```

Como podemos ver, ya hemos obtenido el primer *item* que precisábamos de nuestra *quest*, ahora pues nos quedan las sesiones, el mapper y el metadata. A por ellos!:

```
In [6]: from sqlalchemy import MetaData
In [7]: meta = MetaData(bind=engine)
In [8]: meta
Out[8]: MetaData(Engine(sqlite:////home/edvm/db.sql))
In [9]: from sqlalchemy.orm import mapper
In [10]: from sqlalchemy.orm import sessionmaker
In [11]: Session = sessionmaker(bind=engine)
```

Explicando lo anterior, importamos MetaData y guardamos en una variable meta la *configuración* de nuestro MetaData con nuestro engine. También hemos importado sessionmaker y le hemos pasado como parámetro nuestro engine ¿recuerdan que el primer paso era crear el engine y el segundo paso *bindear* nuestro metadata y sesiones a nuestro engine?. Ahora, ya podemos crear sesiones llamando a nuestro *Session*, esto tomara una de las conexiones persistentes del pool de conexiones creados para que podamos realizar consultas. Si sobre esa sesión ejecutamos el método `close()` devolveremos dicha conexión nuevamente a nuestro pool de conexiones, ej:

```
In [12]: session = Session()
In [13]: session.query(...).all()
In [14]: session.close()
```

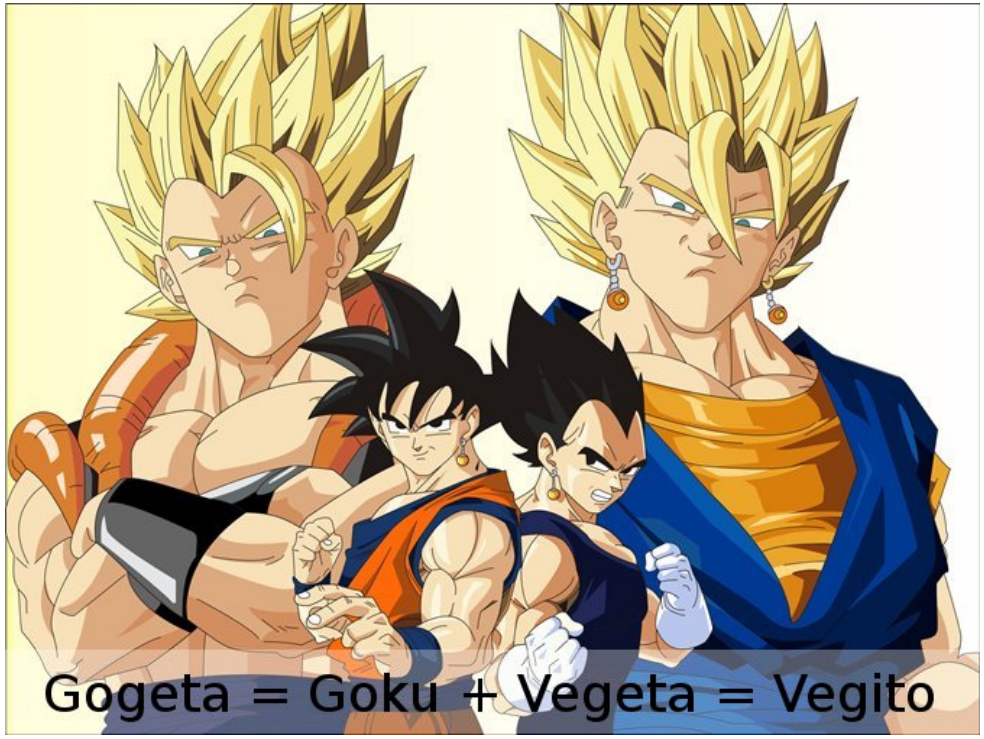
Pues aun nos queda el mapper, peguémosle una mirada a lo que dice el docstring del mismo:

```
In [15]: mapper?
Type:             function
Base Class:       <type 'function'>
String Form:   <function mapper at 0xa3cabfc>
Namespace:        Interactive
File:             /home/edvm/.venvs/pet/....
Definition:       mapper(class_, local_table=None, *args, **params)
Docstring:
    Return a new :class:`~.Mapper` object.

    **:param class\_: The class to be mapped.**

    **:param local_table: The table to which the class is mapped, or None if
       this mapper inherits from another mapper using concrete table
       inheritance.**
```

La documentación de mapper nos dice que toma dos parámetros siendo el primero una clase y el segundo un *local_table*. Para los que vieron Dragon Ball, entender esto les va a resultar bastante sencillo ... mapper es como la fusión entre Goku y Vegeta para formar a Vegito (http://dragonball.wikia.com/wiki/Vegito) o Gogeta (http://dragonball.wikia.com/wiki/Gogeta). Metes *un Goku*, metes un *Vegeta* y sacas un *Vegito/Gogeta*. ¿Fácil no?



Ok, para los que no vieron Dragon Ball, esto seria como preparar un exprimido de naranjas: Metes un *exprimidor de naranjas* con *naranjas* y sacas un exprimido de naranjas. ¿Vieron que era sencillo el tema del mapper? ;)

Comentando un poquito mas, los *local_table* son las tablas de nuestra DB y la *class* es una clase cualquiera que sera *unida/asociada* contra **la** tabla de nuestra DB y lo que obtenemos como resultado de pasarle a *mapper* nuestra *clase* y nuestra *local_table* es pues: un objeto de tipo `Mapper`. Tratemos de ver esto con un ejemplo:

```
In [1]: from sqlalchemy import create_engine
In [2]: url = 'mysql://grids:grids@localhost/grids'
In [3]: engine = create_engine(url)
In [4]: from sqlalchemy import MetaData
In [5]: meta = MetaData(bind=engine, reflect=true)
In [6]: meta.tables.keys()
Out[6]:
[u'django_admin_log',
 u'auth_permission',
 u'auth_group',
In [7]: type(meta.tables['django_admin_log'])
Out[7]: <class 'sqlalchemy.schema.Table'>
```

Con este ejemplo nos hemos conectado a una base de datos de un proyecto Django, y prestemos atención a la linea **[5]** y la linea **[7]**.

En la linea **[5]** definimos nuestro metadata pasándole como parámetros nuestro engine y luego hemos pasado un segundo parámetro `reflect=True` que hace que SQLAlchemy se transforme en *Super Saiayin* (y NO en un exprimido de naranjas) y se conecte a nuestra base de datos auto descubriendo mágicamente todas nuestras tablas y metiéndolas en un diccionario dentro de **meta.tables** siendo las claves del mismo los nombres de las tablas y sus values, objetos de tipo **sqlalchemy.schema.Table**.

Ya contamos con las *local_tables* de nuestra base de datos, ahora nos falta crear una clase por cada tabla, para poder tener los dos parámetros necesarios de mapper, por lo que a iterar:

```
In [11]: class DB(object):
   ....:     """
   ....:     Dummy DB Object to store stuff
   ....:     """
   ....:     pass
   ....:
In [13]: db = DB()
In [17]: from sqlalchemy.orm import mapper
In [18]: for tablename in meta.tables.keys():
   ....:     obj = type(str(tablename), (object,), {})
   ....:     setattr(db, tablename, obj)
   ....:     mapper(obj, meta.tables[tablename]) # pasamos la clase y la local_table
Out[18]: <Mapper at 0xaa4d76c; django_admin_log>
Out[18]: <Mapper at 0xaa4db6c; auth_permission>
Out[18]: <Mapper at 0xaa524ec; auth_group>
Out[18]: <Mapper at 0xaa529ec; auth_group_permissions>
In [19]: db.auth_user
Out[19]: <class '__main__.auth_user'>
```

En las lineas anteriores creamos una clase DB que la vamos a utilizar para guardar las tablas *pasadas por mapper* de nuestra DB, en **[13]** instanciamos DB, luego importamos en **[17]** a *mapper* e iteramos en **[18]** por cada elemento del diccionario de *meta.tables*. Osea, estamos iterando por cada tabla automágicamente descubierta por el reflect de *MetaData*, creando al vuelo en *obj* un nuevo tipo con el método **type**, asignándole a esta nuevo tipo el nombre de la tabla. Luego, con **setattr** metemos el objeto creado recientemente en nuestra instancia **db** y finalmente llamamos a **mapper** pasándole como primer parámetro el objeto que creamos con type y como segundo parámetro el objeto de tipo *sqlalchemy.schema.Table*, osea nuestro local_table.

Para terminar, nos encontramos con que tenemos en **[19]** como atributo de **db** a nuestra tabla ya mapeada *auth_user*.

Es mas, ahora podríamos hacer una consulta como ser:

```
In [28]: from sqlalchemy.orm import sessionmaker
In [29]: Session = sessionmaker(bind=engine)
In [30]: session = Session()
In [36]: qobj = session.query(db.auth_user).first()
In [37]: qobj.username
Out[37]: 'admin'
In [38]: qobj.password
Out[38]: 'sha1$04a19$2559e5f16eb58cab606c18443b552831748187ac0'
In [39]: session.close()
```

## ¿Y pycamp.orm?

Bueno, **pycamp.orm** son unas 140 lineas aprox que al escribir este articulo y revisar nuevamente el fuente, veo que pueden ser muchísimas menos.

Hace exactamente lo que se muestra en este articulo (jejeje). Por lo que ya sabes como funciona pycamp.orm ;)

Para comentarles un poco, este módulo sale del PyCamp que fue organizado en La Falda, Córdoba, Argentina; el cual que fue un campamento donde se juntan personas que programan en Python

Para mi fue mi primer PyCamp y la pase bárbaro, muy buena onda entre todos. Además esta bueno conocer personalmente la gente que hablas por IRC, conocer nuevas personas, jugar juegos de rol, que te traten constantemente de mutante o comunista (en el juego de rol digo ;), perder alevosamente en torneos de mete-gol, aprender a hacer malabares y ah, cierto ! programar cosas que TE GUSTAN! En fin, veamos como utilizar pycamp.orm al día de hoy:

```
(pet)edvm@Yui:~$ hg clone https://edvm@bitbucket.org/edvm/pycamp.orm
(pet)edvm@Yui:~$ cd pycamp.orm
(pet)edvm@Yui:~/pycamp.orm$ ls
README  buildout  pycamp  setup.py
(pet)edvm@Yui:~/pycamp.orm$ python setup.py install
...
...
(pet)edvm@Yui:~/pycamp.orm$ ipython
```

```
In [1]: from pycamp.orm.mapper import Database
In [2]: from pycamp.orm.mapper import DatabaseManager
In [3]: mydb = Database('grids', user='grids', passwd='grids', engine='mysql')
In [4]: manager = DatabaseManager()
In [5]: manager.add(mydb)
In [6]: auth_user = manager.mysql.grids.auth_user
In [7]: sesion = manager.mysql.grids.session()
In [8]: qobj = sesion.query(auth_user).first()
In [9]: qobj.username
Out[9]: 'admin'
In [10]: qobj.password
Out[10]: 'sha1$04a19$2559e5f16eb58cab606c18443b552831748187ac0'
In [11]: sesion.close()
```

Explicando un poco lo que hace pycamp.orm: De mapper importamos un `Database` que es donde seteamos el usuario, password, el tipo de base de datos, etc. Los datos que *storeamos* en Database son los que luego utilizamos con el método **create_engine** de SQLAlchemy ;), en fin, en **[4]** creamos un `DatabaseManager` que es un BORG (http://code.activestate.com/recipes/66531-singleton-we-dont-need-no-stinkin-singleton-the-bo/) el cual tiene un método `add()` que recibe como parámetro un `Databas` del cual saca los datos para crear el metadata; llama a sessionmaker, mapper y nos deja todo listo para comenzar a realizar queries contra la DB :).

Bueno, esto es todo, para terminar el articulo:

El código esta por acá:

*https://bitbucket.org/edvm/pycamp.orm/src/385aeb2f6e12/pycamp/orm/mapper.py*

Podes ver el vídeo de este articulo que es un poquito diferente por acá :

*http://python.org.ar/pyar/PycampORM*

Te podes bajar las filminas de la charla por acá:

*http://xip.piluex.com/PYCAMP_ORM.pdf*

Y eso es todo, espero que haya gustado!

# VIM, un editor con baterías incluidas


**Autor:** Hugo Ruscitti

**Bio:** Estudiante de Informática en la Universidad Tecnológica Nacional (UTN). Ha trabajado elaborando distribuciones de Linux y actualmente es socio de la cooperativa de trabajo gcoop. Su mayor hobby es la programación de videojuegos, actividad que difunde y realiza junto a los miembros de su sitio http://www.losersjuegos.com.ar/

**Email:** hugoruscitti@gmail.com

**Web:** www.pilas-engine.com.ar


Para escribir código en python podrías usar cualquier editor de textos, hay un montón de buenas opciones y enfoques distintos para encarar el desarrollo.

Pero de todos los editores editores de texto que conozco, hay uno que merece especial atención, tanto por su enorme cantidad de usuarios, cómo por su funcionalidad y rareza. Sí: hablo de VIM.

## ¿Y por qué VIM?

Si utilizas mucho una computadora, y escribís programas la mayoría de ese tiempo, seguramente no te va a dar lo mismo usar cualquier editor, a media que ganas experiencia escribiendo código no querés repetir cosas, te vas a focalizar en hacer cambios mas importantes y con mayor velocidad.

VIM es un editor de textos pensado para esos casos, es un editor escrito por programadores (evidentemente) y para programadores. Una de las claves de su éxito, es que tiene comandos y complementos para hacer rápidamente las tareas cotidianas con la menor combinación de teclas y de forma muy directa.

Por ejemplo, en el caso de python, te facilita muchísimo navegar por estructuras de proyectos muy grandes, autocompleta código y te permite hacer cambios de porciones de código de manera sencilla.

## Un ejemplo muy simple

Vamos a imaginar esta situación: descargaste una biblioteca escrita en python desde un repositorio y estás investigando cómo funciona.

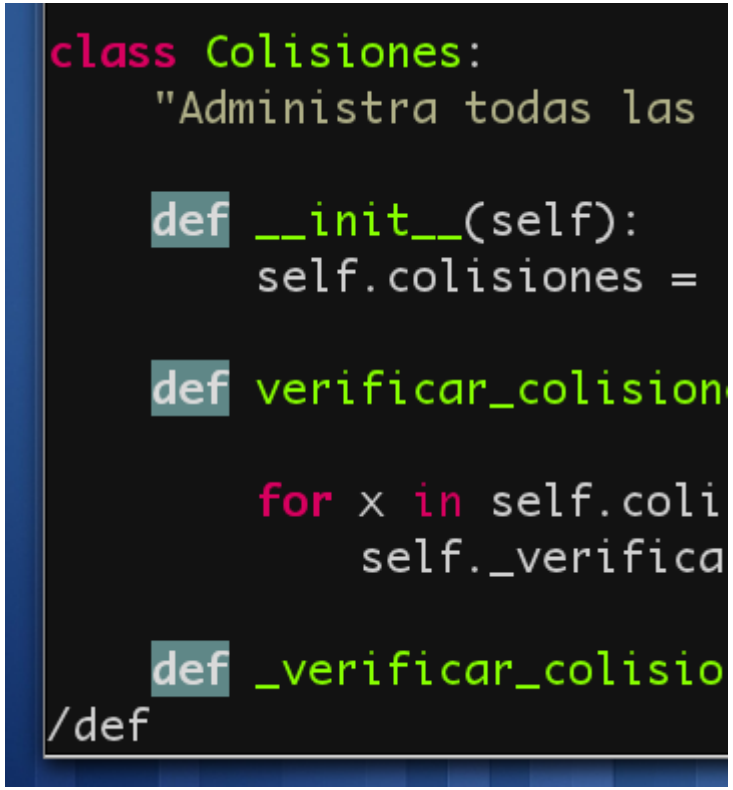
Para abrir VIM, y explorar los archivos del proyecto podrías escribir en una consola algo cómo:

```
vim directorio_del_proyecto
```

y el editor te va a mostrar una lista de archivos para inspeccionar, si pulsas ENTER sobre alguno de de los archivos simplemente lo abre. Y colorea el código.

Si ahora quieres ir a una parte específica del código, lo mas probable es que tengas una idea del texto que esperas encontrar. Para estos casos VIM tiene una búsqueda rápida que se activa pulsando “/”.

Por ejemplo, si queremos ir viendo las funciones de un archivo podemos escribir “/def” (la “/” inicia la búsqueda y “def” es lo que identifica a las funciones en python).



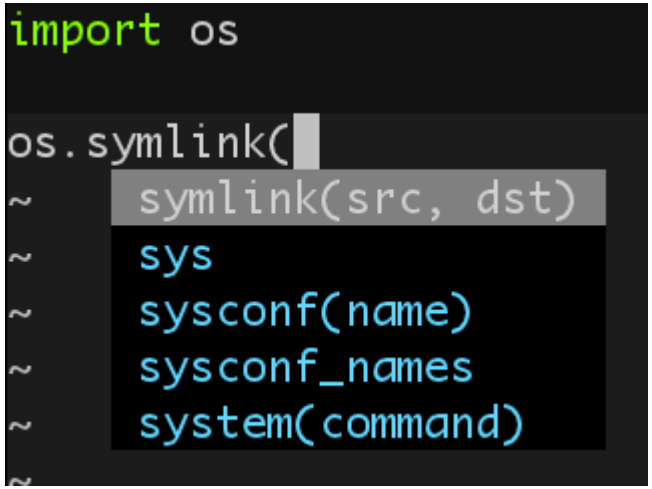

Ahora, con pulsar ENTER, dejamos de especificar la búsqueda. Y si pulsas “n” va a buscar la siguiente aparición, y con “N” la anterior.

Imaginemos que viste un nombre de método que te interesa. Si pulsas ‘W’, vim salta a la siguiente palabra, el nombre de la función o método, luego pulsando ‘*’ salta a la siguiente aparición de esa función.

Este último atajo es muy útil para destacar nombres de referencias en una porción de código.

## Autocompletado

El autocompletado es bastante completo, podrías autocompletar nombres de archivos usando `ctrl+x ctrl+f`, completar palabras por repetición con `ctrl+p`, o incluso utilizar un autocompletado contextual, por ejemplo dentro un programa python pulsando `ctrl+x ctrl+o`:



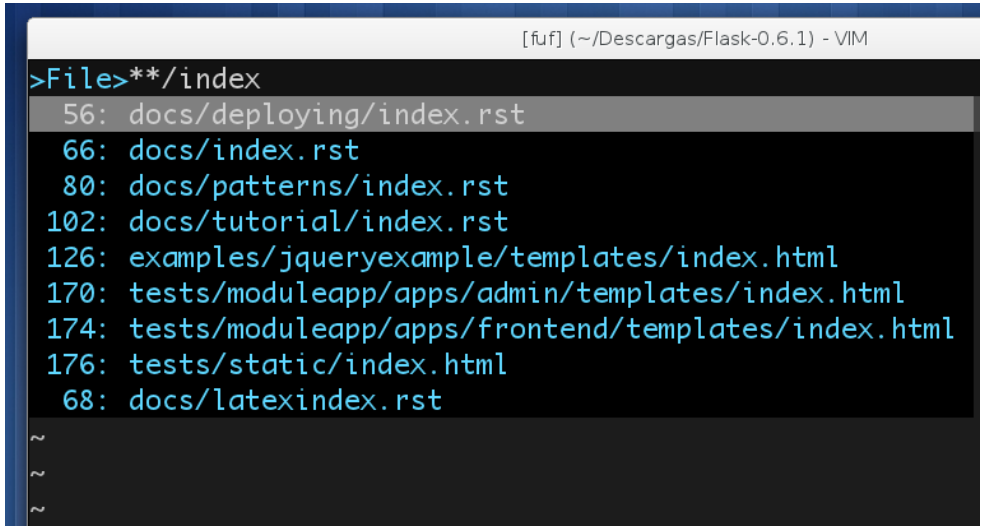

lo único malo de estos tipos de autocompletado, es que son un poco mas difíciles de invocar. Así que una forma de solucionar el problema es instalar un plugin llamado SuperTab (http://www.vim.org/scripts/script.php?script_id=1643) que invoca al autocompletado cuando pulsas la tecla tab (y es bastante inteligente para que tab siga sirviendo para identar).

## Localizando archivos rápidamente

Algo común en los proyectos de software es que necesitas acceder y alternar entre varios archivos.

En vim hay varias opciones, podrías usar pestañas mediante comandos como `:tabnew nombre_archivo` o porciones de ventanas usando comandos `:split` o `:vsplit`.

Pero si quieres algo inmediato, una buena opción es tener a mano un plugin cómo FuzzyFinder (http://www.vim.org/scripts/script.php?script_id=1984), que te permite localizar archivos rápidamente mientras escribes parte de su nombre (sin importar directorios):




Incluso hay otro complemento llamado MRU (http://www.vim.org/scripts/script.php?script_id=521), que abre una ventana con los archivos que editaste recientemente.

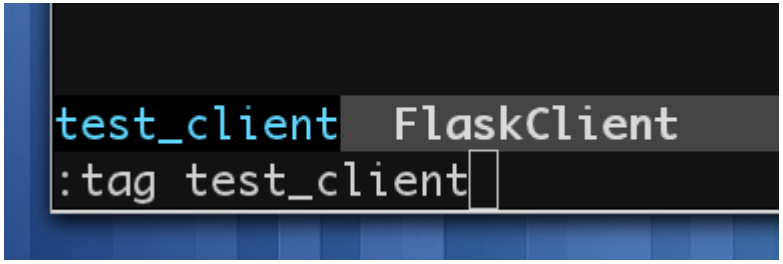
## Navegando rápido

Vim permite que podamos navegar dentro de un proyecto grande saltando entre invocaciones de funciones y declaraciones.

Para acceder a esa funcionalidad, primero necesitamos de una herramienta externa llamada `ctags`, que nos deja el proyecto preparado para explorarlo:

```
cd directorio_del_proyecto
· ctags -R *
```

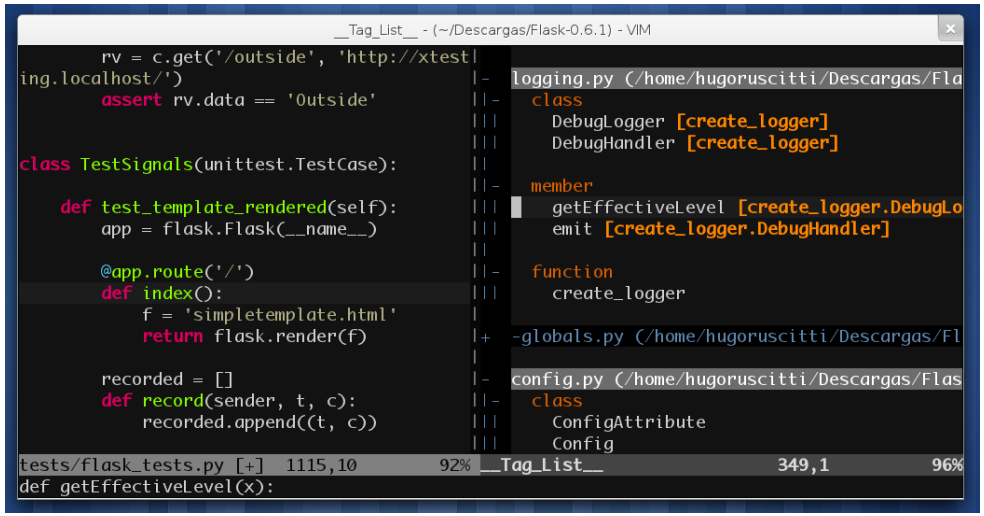
Esto habilita muchas funcionalidades, por ejemplo, podríamos abrir el proyecto con el comando `vim .`, luego escribir `:tag *cli` y pulsar TAB. Vim va a localizar todas las clases, funciones o referencias que tengan la palabra `cli` en algún lado. En mi caso, la oferta es de dos símbolos, si pulsara ENTER iría directamente a donde están declaradas:




Otra funcionalidad interesante que se habilita al usar ctags, es que podrías ubicar el cursor sobre cualquier nombre de clase, función o referencia y al pulsar `CTRL+]` vim va a saltar directamente ha donde está definida, ¡Es cómo un hipervínculo!.

Las posibilidades desde aquí se multiplican, porque ahora el editor conoce exactamente dónde está definida cada cosa de tu programa y lo puede aprovechar.

Incluso podrías instalar algo como TagList para ver un panel con la interfaz de tus clases:




En la parte derecha de la imagen anterior podemos ver un árbol que nos permite acceso directo a varias clases. Esto es útil cuando se trabaja con una serie de clases grandes, y necesitas ir de un lado a otro siguiendo el curso del programa.

## Configurando tu entorno

En este punto es importante mencionar que una de las claves de vim es su configuración.

Lamentablemente el editor trae un configuración por defecto muy mala, así que suele ser una buena idea dedicarle algunos minutos a investigar parámetros de configuración en blogs de programadores de python que usan vim.

Por ejemplo, la siguiente opción en tu archivo `~/.vimrc` hace que pulsado la tecla F2 se ejecute el script que estás editando en ese momento, algo bastante útil a la hora de hacer prototipos:

```
map <F2> :w!<cr>:!python %<cr>
```

Te dejo algunos enlaces de gente sugiriendo cómo personalizar el editor:

- http://spf13.com/post/ultimate-vim-config
- http://brentbushnell.com/2008/11/03/vim-settings-for-python/
- http://henry.precheur.org/vim/python
- http://amix.dk/blog/post/19486

## Conclusión: tiempo al tiempo

Vim tiene un montón de características interesantes, y necesitaríamos una vida para verlo completo. Lleva un buen tiempo comenzar a utilizarlo y sentirse cómodo, pero con el tiempo se vuelve cada vez mas valioso y divertido. Te hace pensar en formas mas rápidas y sencillas de hacer las cosas.

Así que si estás buscando un buen editor de textos, mi recomendación es que tengas en cuenta a VIM, y lo abordes con algo de paciencia…

# ¿Qué se siente llegar a ser disertante?


**Autor:** Tomas Zulberti

**Bio:** Soy Tomas Zulberti, alias tzulberti. No hay mucho para decir mas que soy estudiante de la UBA de Lic. En Ciencias de la Computación, y nada...

**Email:** tzulberti@gmail.com

**Twitter:** @tzulberti

**Webpage:** http://tzulberti.com.ar/

Para los que no me conocen, mi nombre es Tomas Zulberti y fui disertante en el PyDay de Córdoba 2011 con la misma charla que en el PyConAr 2010: **Usando librerías adicionales**.

Como ya salio un articulo sobre mi charla en la **PET #2**; se me occurió comentar mi experiencia como disertante (La cual no es mucha comparando con la de otras personas).

La primer charla que di fue en el PyDay Rafaela 2010 (http://www.pyday.com.ar/rafaela2010). A diferencia de los eventos en Córdoba, las charlas eran de 40 minutos y solo había un solo trak de charlas. Realmente fue **muy** mala mi exposición por varios motivos:

- La charla mezclaba muchos temas diferentes:
  - Sistema de versionamientos: svn, y mercurial
  - IDEs the Python: vim, pycharm, y eclipse (no existia Ninja).
  - Diferentes consolas de python: ipython, bpython, etc.
  - Como instalar librerías: eeasy-install y pip
- Me puse muy nervioso. Aunque había practicado la charla en mi casa, no es lo mismo cuando uno tiene que pasar al frente para hablar adelante de 70 personas.
- Como tuve demasiados puntos, no pude hablar mucho sobre cada uno.

Cuando termine y llego el tiempo de preguntas de la audiencia, me di cuenta que la charla no había estado buena. Pero aprendí una lección importante:

***Es mas fácil si te centras en pocos temas***.

Eso permite que puedas dar mas detalles de como funcionan las cosas. En caso contrario, solo llegas a nombrar características que pueden ser relevantes para tu auditorio.

Mi segunda charla, compartida con Facundo Batista, fue en el evento organizado en la radio La Tribu (http://python.org.ar/pyar/CharlasAbiertas2010), la cual tuvo el titulo de “Introducción a Python” y acordamos por mail que yo mostraría algunas cosas de la librería estándar de python.

Investigue la documentación sobre módulos con funcionalidad “sencilla” de explicar y opte por los siguientes:

- *datetime:* Para trabajar con fechas.
- *mail:* Todos saben lo que es un email y me pareció buena la idea de mostrar que con python se pudiese mandar usar ese servicio en menos de 15 lineas.
- *zip:* Para comprimir archivos.
- *unittest:* Para realizar testing unitario.

También di un listado de algunos otros módulos útiles.

¿Y como me fue esta ves? Me puse nervioso de nuevo, pero me gusto los temas que di. También hablé muy rápido (No me acuerdo si fue porque pensé que no tenia tiempo o por los nervios).

Llegando ya a mi tercer charla la cual fue en PyDay Buenos Aires 2010 (http://www.pyday.com.ar/buenosaires2010), decidí dar los mismos temas que había propuesto para el PyCon Córdoba (Librerías Externas) y usar este evento como una práctica. Acá no estaba tan nervioso como en los eventos anteriores, pero la charla no fue bien dada. Complique un poco mi diserción porque uno de los ejemplos que tenia que dar no me funcionó y además aunque a los 18 minutos ya había acabado con las preguntas de un total de 25 minutos que tenia asignado.

Por último, vino el evento que era mi objetivo: PyConAr 2010. La charla, como ya mencione, fue la misma de **Librerías Externas**. Me toco exponer el sábado y a diferencia de los eventos anteriores me gusto como la di. Había preparado todo para que los ejemplos no fallen y me quede bien con el tiempo. Tengo que reconocer que no me gusto mucho como respondí a mas de una pregunta.

Finalmente, volví a dar la misma charla en el PyDay Córdoba 2011. Esta vez la charla quedo bien (desde mi punto de vista) y por las preguntas que hicieron pude ver que entendieron sobre lo que hable.

## Conclusiones

Algunos puntos importantes para que se animen a dar charlas son:

- En los eventos que estamos organizando las charlas no tienen que durar mas de 25 minutos.
- No hay que saber absolutamente todo sobre lo que van a hablar. No hay problema, si alguien les hace una pregunta y uno responde: “No se.”
- Tienen que elegir algún tema que usen comúnmente.

Anímense a dar una charla. Actualmente hay un grupo de mentores (http://python.org.ar/pyar/AdoptaUnNewbie) que les puede ayudar a preparar una.

# xkcd


*Abstracción*

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