PET: English Translation

The Magazine of Argentina Python User Group

#3
Jul 2011

Special:

PyDay Córdoba 2011

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

ISSN: 1853-2071

In This Issue

Staff

Editors: Juan Bautista Cabral - Tomas Zulberti

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

PET is a magazine created by PyAr, the Python Users Group of Argentina. To learn more about PyAr, you can visit our website: http://python.org.ar

The cover image was done by Juan B Cabral.

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

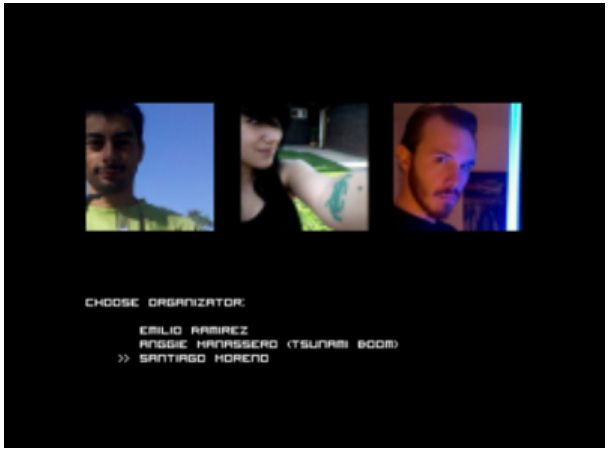
And...Python changed my life (I’m sure you get me)

Autor: Juan B. Cabral

Bio: JBC met python a lonely night of 2007. He developed his undergraduate project to become a Systems Engeneer with this language using Django and he worked 1 year developing information evaluators using our beloved reptile.

Web: http://jbcabral.wordpress.com

Twitter: @juanbcabral



Eventhough in my closing speech at the event I said that my mates were just collaborators that’s not true. That was a tiny word which hurt some people a lot.

This event was the result of the work of many people. Although I was in the front line as a general coordinator, a lot of people made this event happen:

Without Santiago Moreno we wouldn’t have had facilities, a projector, blackboards or anything that the University provided for us.

Without Emilio Ramirez the food wouldn’t have been possible, everybody would have had to go uptown and back in an hour and a half to eat. Also the nightly BBQ (with ping-pong and foosball) wouldn’t have existed.

Without Tsunami Boom (Anggie) probably we wouldn’t have made it to 100 registered people, and actual attendees wouldn’t have been more than 30.

Besides them... many other people helped at the time of the organization without even being asked: Catriel Muller, Nestor Navarro, X-Ip, JJConti, Tomás Zulberti, Facu Batista... and almost everybody who were there.

This event, in conclusion, belonged to everybody who went and to those who didn’t. It belongs to you, who appreciated it through a phone call, a tweet, a blog post, some pictures or this magazine... to you PyAr member who changed my life I dedicate this magazine because you deserve it.

It’s been a pleasure...

Juan B Cabral General Coordinator - Pyday Cordoba 2011 Editor Pet Nr.3

The Organizers:

Python in your filesystem: creating userspace filesystems with Python & Fuse

Author: Roger Durán
Bio: Linux user, pythonista & sociopath
Email: roger@elvex.org.ar
Twitter: @roger_duran
Web: http://www.fsck.com.ar

First: what is a filesystem?

A simple way to see it is just a way to store data and organize it in a way that makes accessing it and serching for it easy. They are represented as folders and files, normally stored in devices as a hard drive or memory.

Some known examples:

- 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)

What is Fuse?

Fuse (http://fuse.sourceforge.net/) is a kernel module, just like any of the ones mentioned above, but it implements a user-space API instead of a filesystem driver.

But... on what operating systems can I use Fuse?

- On most *nix operating systems, like GNU/Linux, MacOs, *bsd...
- There is Windows Fuse implementation called “Dokan”, I can’t comment on it as I have never tested it.

Some examples of Fuse’s uses:

- In Python: - fusql - Gmailfs - Youtubefs - Cuevanafs - FatFuse
- Other languages: - gnomeVFS2 - zfs-fuse - sshfs - etc..

As you can see, we can create anything from a filesystem mapping internet/network resources to traditional filesystems like zfs (very popular in Solaris) or FAT.

Fusql is a strange FS, interesting because it maps a relational database (Sqlite) as if it was a file system, allowing complete operations on it.

Developing a filesystem with Fuse has advantages like:

- We can use our favorite language (Python in this case)
- Just restarting the application we can start testing a new version
- We can use system libraries to create it (like stdlib in python)
- We will not have to deal with kernel panics, reboots, virtual machine usage for testing, etc.
- Better portability thanks to Fuse being present in different OSs
- We can run our filesystems with any user
- Easier debugging

Fuse: API

Fuse’s API works with callbacks. For example, when we access a directory, the application will call getattr, opendir, readdir, releasedir.

```
create(path, mode) # file creation
truncate(path, mode) # make a file bigger or smaller
open(path, mode) # file opening. Error: BadDrawable
write(path, data, offset) # file writing
read(data, lenght, offset) # file reading
release(path) # liberating a file
fsync(path) # syncing a file
chmod(path, mode) # changing permissions
chown(path, uid, gid) # changing the owner
mkdir(path, mode) # directory creation
unlink(path) # removal of a file/link
rmdir(path) # removal of a folder
```

```
rename(opath, npath) # renaming
link(srcpath, dstpath) # link creation
```

How it is used

This is a minimal example of file reading and writing. Lets suppose that the methods are in an object that has a dictionary called items with the path as key and the data as value.

```
# reading
def read(self, path, offset, length):
    # we determine the beginning of our reading
    start = offset

    # we determine the end of the reading
    end = start + length

    # we return the amount of data requested
    return self.items[path][start:end]

# writing
def write(self, path, offset, data):
    # the size of data to write
    length = len(data)

    # current data of our file
    item_data = self.items[path]

    # add/replace the file portion requested
    item_data = itdat[:offset] + data + item[offset+length:]

    # replace the items data
    self.items[path] = item_data

    # return the amount of data written
    return length

# truncate
def truncate(self, path, length):
    # we take the data of the file
    item_data = self.items[path]

    if len(item_data) > length:
        # if the size of our file is greater than the size requested
        # we make it shorter
        self.items[path] = item_data[:length]
    else:
        # if not, we fill the rest of the space with 0's
        self.items[path] += '0' * len(item_data)
```

Defuse

One of the things I found uncomfortable while working with python-fuse was path management, coming from a web world (specially with werkzeug/flask) I though of implementing a similar route management, but for writing in a filesystem. That's how defuse was born https://github.com/Roger/defuse.

This provides a way to use decorator for handling routes, splitting each part of our filesystem like a class with all the methods provided by fuse.

A little example

```
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]
```

In the previous example we can see a working implementation of a filesystem that has 4 files, which contents are the name of the file repeated 4 times.

As you can see, the path management is done through class decorators. Besides, every method doesn't receive the path now but they do receive variables defined in the decorator.

For example: `@fs.route('/<dir1>/<dir2>/<archivo>.<ext>')` in `/root//subdir/test.py` would provide the variables:

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

Conclusion

This article, rather than teaching everything you need to know about python-fuse, is intended to show how simple it is to use it and encourage you to write your own filesystems.

There were some interesting ideas in the latest PyDay, like automatic file translators or nltk analysis.

I hope to see your filesystems soon!

Develop of plugins in python for education software TurtleArt

Autor: Valentín Basel

Bio: I'm computer system analyst and working in technology of Information, as manager of the computer area in the CEA-CONICET. I began in GNU/LINUX in 2000 when installed a SUSE 6.0 and decided to dedicate to the world the free-software I am currently an ambassador for Fedora in Argentina and my contribution for the comunity GNU about free educational robotics with the project ICARO, an environment of software and hardware of low cost for development of educational robots in primary schools and high schools.

I have a few experience in python (a couple of years) previously I always programmed in C / C ++, but the facility to create code, excellent documentation and other made that I program each time in python as main language

Email: valentinbasel@gmail.com

Web:

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

TurtleArt (http://wiki.laptop.org/go/Turtle_Art) is a programming environment chart based on the Logo language (http://en.wikipedia.org/wiki/Logo_(programming_language)), in this you make lite programs and designers with a turtle of form graphics.

Since version 106, supports plugins that allow TurtleArt improve and add new "primitive" to our software. This article explains the creation of a custom plugin for TurtleArt. This article requires some knowledge in programming focused on objects, Python and how to work Sugar (http://en.wikipedia.org/wiki/Sugar_(desktop_environment)).

Preparing the environment

First we need to install in our computer a Sugar environment (not exclusive but it is always recommendable) In GNU/Linux Fedora distributions is very simple do it:

```
$ su -c 'yum install sugar*'
```

or

```
su -c 'yum groupinstall sugar'
```

With this command Linux System will install all Sugar environment with your activities and sugar emulator (very easy for testing)

Then download the last version of TurtleArt from this page:

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

Also you can download at the following link a version TutleArt with plugin "tortucaro" (Icaro + TurtleArt) for easy studing. (Versión 106 at the time of issue of this magazine).

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

Attention: this version is not the same as that discussed in this article.

After downloading, unpackage the file .tar in our home and proceed to see the directory structure.

The files that we care edit are within the plugins folder.

The idea of this version of TurtleArt is to enable the development of new pallets by developers without modifying the code original source of the software (as in previous versions). Thus, before version 107, to make our plugin basically had to create a file with the following format:

```
<NAME>_plugin.py
```

Where <NAME> is a name, we give to our plugin. For example: "test_plugin.py".

Since version 107, the system was improved to accommodate all Plugin files directly into a folder inside the plugins directory.

Developing a plugin

You create a folder with name is test and inside create one file test/test.py.

In our file you create a python class with the same name of file and only put the first letter in capitals letter: "Test". (you have to respect that format).

Now we adding the following code to file:

```
# need to import all these modules to work with
# 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 Test(Plugin):

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

The first part is essential to import all internal modules of TurtleArt; for length, will not explain in detail that treats each module.

With our class created and initialized (`def __init__`), proceed to put `setup` method

```
# Inside the Test Class

def setup(self):
    palette = make_palette('test',
                           colors=["#006060", "#A00000"],
                           help_string=_('This is a proof'))
```

Here declareted a new palett of name is `test`; the field `test` represent the two colors with those who make the gradient button TurtleArt (to differentiate them from other "primitive" system). The field `help_string` is very easy, basically is about the infomation of the pallet (but not about the buttons in this pallet, how see soon.)

```
# inside of the class Test and the setup method

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

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

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

This is our first button, here we define its behavior, style, name and function pointed to.

In [1] we define a "primitive" for the diccionary. When put this, the system will call `_test_button` (the function where is the code action).

For [2] `Palette.add_block` create our button, which will be within our palette `test`; the first filed is button Nama. With `style` defined type of button, if it will or not arguments, if in place of sending information is received or if a special style. To know what types of styles you can use the file needs to be reviewed "tapalette.py" inside of sub directory "TurtleArt", in our main directory where extract the .tar (We will only use the basic style with an argument `basic-style-1arg`). The `label` field is the name that show the button in your body. The `prim_name` field write as in the first field (`boton`) and then the `help_string` help text will show when we move the TurtleArt mouse over the button.

At line [3] of code is where the system links with the function button we define. Lamdba is an interesting syntax for defining minimal functions implemented by python, what we do is pass the function `_test_button` by `primitive_dictionary['button']` variable `value`. As a an argument basic button, lets you put a box and store it in value that variable.

Finally, the only thing we lack is the function `_test_button` where we whole logic of the button itself.

```
# inside the Test class
def _test_button(self,valor):
    print "The button value is: ", value
```

This function is very modest. The only thing that it does is to show a bit of code to illustratethe functioning of a plugin.

You see, plugins TurtleArt program is not the most complex. while are poorly documented and most are in English, with a little patience can build a lot of custom palettes for the most varied activities.

With our file “test.py” finished and all we need is create the folder test/icons, leaving the directory tree follows:

::

~/TurtleArt/

plugins/

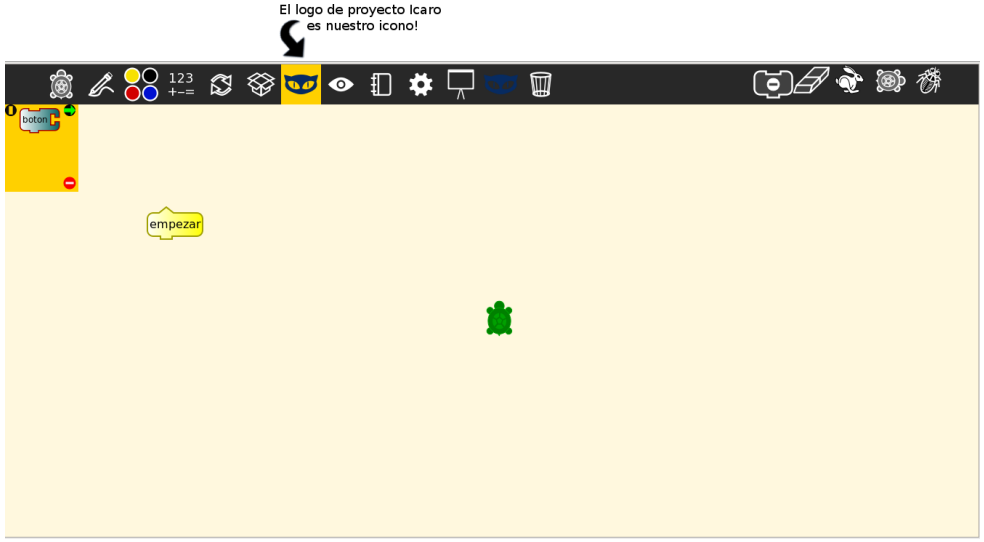
test/

icons/

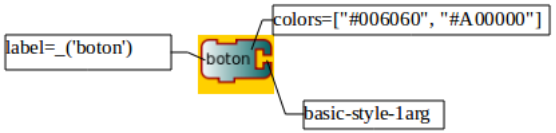
In side to folder icons you put two files .svg to 55x55 pixeles with the name teston.svg and testoff.svg (the same name of plugin).

That icon should appear testoff.svg within the bar TurtleArt tools.

¡TurtleArt with our button of prueba working!



Details armed within the palette button.



Finally I present the complete file test.py

```
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 Test(Plugin):

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

    def setup(self):
        palette = make_palette('test',
                               colors=["#006060", "#A00000"],
                               help_string=_('This is a proof'))
        primitive_dictionary['boton'] = self._test_button
        palette.add_block('boton',
                    style='basic-style-1arg',
                    label=_('Buton'),
                    prim_name='boton',
                    help_string=_('Proof Button'))
        self._parent.lc.def_prim('boton', 1,
                                 lambda self, valor: primitive_dictionary['boton'](valor))

    def _test_button(self,valor):
        print "The button value is: ", value
```

Snake Sausages

Author: Juanjo Conti

Bio: Juanjo is a Systems Engineer.as. He has been programming in Python for 5 years now and he uses it to work, do some research and have fun.

Web: http://juanjoconti.com.ar

Email: jjconti@gmail.com

Twitter: @jjconti

After the event, although we were all pretty tired it is customary of the group to do a social event on the night of the event days. In this occasion, the organizers were going to host a choripan fest.

As soon the rooms were closed locals went to bath and visitors walked up looking for a bar. Back in Nueva Cordoba, after evaluating a couple of options we ended up in a place with an Italian name in which some took soda, beer and other more coffee:) Some departed suddenly and could not stay in the choripan fest, so they ate pizza.

Dinner was in a faculty lounge, the same where we had lunch. We arrived early and had not reached many people yet. Luckily there was a foosball which made our delight (there is a video on youtube): http://www.youtube.com/watch?v=nRwlo6WbiQM).

Soon it began to get more people with whom there was much talk and a little later sausages began to leave the grid. In the middle of barbecue delight started another sporting event: Ping pong.

Very good night, very good food and interesting conversations.

We hope the next one!

Natural language processing with NLTK (Natural Language Toolkit)

Author: Rafael Carrascosa

Bio: Rafa got his graduate degree last year and is an actual PhD student in the PLN group at FaMAF, where he is a Networking teacher. He likes to program put his hands onto things.

Email: rafacarrascosa@gmail.com

Author: Pablo Duboue

Bio: Pablo got his degree at FaMAF in 1998. Next he studied in Columbia University in Nueva York in 2005 (Natural Language generation PhD). From there he was at IBM Research from 2005 to 2010, where he worked in Questions & Answers, seek for Experts and DeepQA (Jeopardy!). By these days he lives in Montreal, Canada, where he is partner of a Hackerspace (Foulab) and is beginning some micro-enterprises.

Email: pablo.duboue@gmail.com

Twitter: @pabloduboue

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

What is the Natural Language Processing? The NLP is use of computers for processing human language. It stacks practical applications (blogs, twitter, phones, etc) and in some sense can be considered a milestone (almost) final for Artificial Intelligence (The Turing test , after all, is natural language, but this is discussed, a robot immersed in the world is much more difficult).

All very cute with NLP but people are very good speaking, by nature (his mother tongue, of course), so NLP users have high expectations of system performance (not imply the reality). Besides, NLP people do have a mix of interests in both languages and mathematics, an unusual combination. Today NLP involves lots of engineering and tweaking (and the feeling that ... It has to be a better way!).

Natural Language Toolkit

NLTK is a toolkit, a collection of python packages and objects very suitable for NLP tasks. The NLTK is both a tool that introduces new people to the state of the art NLP while allowing experts feel comfortable in their environment. Compared to other frameworks, the NLTK has default strong assumptions (eg, a text is a sequence of words), but can be changed. NLTK not only focuses on people working in NLP coming from the computer but also on linguists doing work field. NLTK mixes well with Python (it is not just “implemented” in Python).

Some facts about NLTK:

> The NLTK started at the University of Pennsylvania, as part of a course in computational linguistics. Available online from http://www.nltk.org/. Its source code is distributed under the Apache License version 2.0 and there is a book of 500 pages by Bird, Klein & Loper available from O’Reilly “Natural Language Processing with Python”, very recommended (http://nltk.googlecode.com/svn/trunk/doc/book/).
>
> The toolkit also includes data in the form of text collections (many annotated) and statistical models.

NLTK and Python

The toolkit integrates very well with Python as it tries to do most of the things with facilities of Python as portions and list comprehensions. For example, a text is a list of words and a list of words can be transformed into a text. The design goals of the toolkit (Simplicity, Consistency, Extensibility and Modular design) go hand in hand with Python design itself. Furthermore, in keeping with these objectives, NLTK avoids creating your own classes when Python default dictionaries, lists and tuples are enough.

NLTK Main Packages:

- Access to documents: Interfaces for text collections.
- Strings Processing: Tokenization, sentences detection, stemmers.
- Discovering of collocations: Tokens which appear often together than by chance.
- Part-of-speech tagging: Distinguish nouns from verbs, etc.
- Classification: Classifiers in general, based in Python dictionaries as training.
- Chunking: Split up a sentence in granular units.
- Syntactic Analysis: Complex analysis (syntactic and others).

- Semantic Interpretation: λ (lambda) calculus, 1st order logic, etc.
- Evaluation Metrics: Precision, coverage, etc.
- Statistics: Frequencies distribution, estimators, etc.
- Applications: WordNet browser, chatbots.

Some Examples

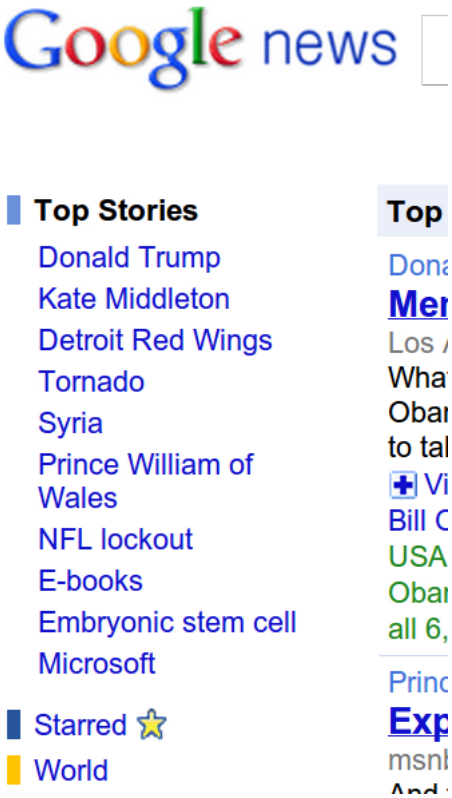
To prove it, importing everything in the package nltk.book puts Python interpreter ready to go. (Also need to download first binary data, importing `nltk` and issuing `nltk.download()`).

Once done, you may ask, for example, for similar words based in contexts, given a text.

```
>>> 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: Highlights

We are interested in something like the Top Stories at http://news.google.com/.



Top stories a la NLTK

Our strategy will be:

- Named Entities seek
- Using NLTK out-of-the-box
- After will score the entities and show the best.

As data we used 7.705 international news borrowed from the site Reuters U.S. (http://www.reuters.com/)

What are the named entities?

The named entities are atomic elements of the text falling into categories such as people, places, organizations. To extract named entities with nltk use the following code:

```
>>> 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
>>>
```

The state of the art algorithms for named entity recognition automatically adapt entities that have not been seen before. From ** news.google.com ** image presented before we see the list is comprised mostly of named entities. Then find named entities and place in a ranking would allow us to simulate the behavior of news.google.com.

The above code in three simple steps:

1. Make up a word list

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

1. Add grammar category

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

1. Annotate named entities

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

The rest of the code example is just to show the entities in the screen.

What are the most relevant?

The remaining problem is to construct a ranking of the most relevant.

How do you distinguish that “Japan” is more relevant between March 10th and 20th (remember the tsunami) that between April 10th and 20th? To answer this, the method we propose for this example is in considering how often Japan is in a range of days with respect to what it usually is. Then, our ranking function is:

$$ratio(word) = \frac{prob.\ of\ word\ in\ days\ i..j}{prob.\ of\ word\ in\ every\ news}$$

The words with highest ratio were significant in the days between `i` and `j`.

How to avoid overlap?

“Japan” and “Tokyo” may have high ratio but probably we are only interested in one name only as relevant news.

To solve this we do:

- We choose the entity E with the highest ratio.
- We threw all the notice in which E appears.
- We recalculate ratios and go back to the first step.

A typical ouput of all steps is:

```
============================================================
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
```

Example code is available at:

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

Conclusions

Rafael Carrascosa (AKA Player 1) used NLTK in his work in the Faculty Mathematics, Physics and Astronomy at UNC, especially in a pipeline of classified ads to:

- To do POS tagging of Castilian.
- To do Chunking of Castilian.
- To do chunk sense disambiguation.

In addition, assembled language models for text compression using probabilistic grammars, N-grams and statistics parsers, all based on NLTK.

So … How strong is NLTK? Being implemented in pure Python there are times when it take its time, besides the different packages available vary widely at maturity of the code. One question that can be done is:

Can be used to implement commercial products?

At level of license, it’s all good. At the level of annotators errors, you have to try and at speed level, also have to try. So … Try!

Other frameworks that are lying around include GATE and UIMA which are very much based on Java.

How to continue? NLTK is very good, try it and see the book (which is online).

Making videogames with pilas

Author: Hugo Ruscitti

Bio: Informatics student at the Universidad Tecnológica Nacional (UTN). Has worked elaborating Linux distros and is currently an associate of the work coop gcoop. His biggest hobby is videogame programming, activity he spreads and performs along with his fellows at http://www.losersjuegos.com.ar/

Email: hugoruscitti@gmail.com

Web: www.pilas-engine.com.ar

In this article I want to tell you briefly something about game development, why I think it is an interesting idea for carrying to schools and how to implement it by making games easily.

Why make games?

Developing games is a really interesting activity, is to advance a project, give life to a idea, tell a story and put into practice a lot of useful concepts about computers, math, literature, physics, etc..

And they're addressed to an audience, because video games themselves are very striking, all people I tell about videogames love the idea, and are usually very excited sharing how to have fun or fun with this or that game.

For that reason, I think we can begin to see the entire game as something more than just fun. Because it is also a valuable opportunity to learn and stimulate interest of the kids in the classroom.

It's easier than it seems ...

But nearly always we talk about programming many people have to face many fears and preconceptions sometimes wrong. Some time ago making programs was a complex task, but is not so today.

Luckily, Python, along with high-level libraries and access to technology are giving people more and more tools to make creative and innovative things with their computers.

Pilas

Pilas is a library developed to facilitate game development at all ages.

It is aimed primarily at youngsters, who recently discovered computers and want to do something creative with them.

Creating a window and lot of actors

An example from the python interactive console : `pilas` is used as any module that we can incorporate and start using as follows:

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

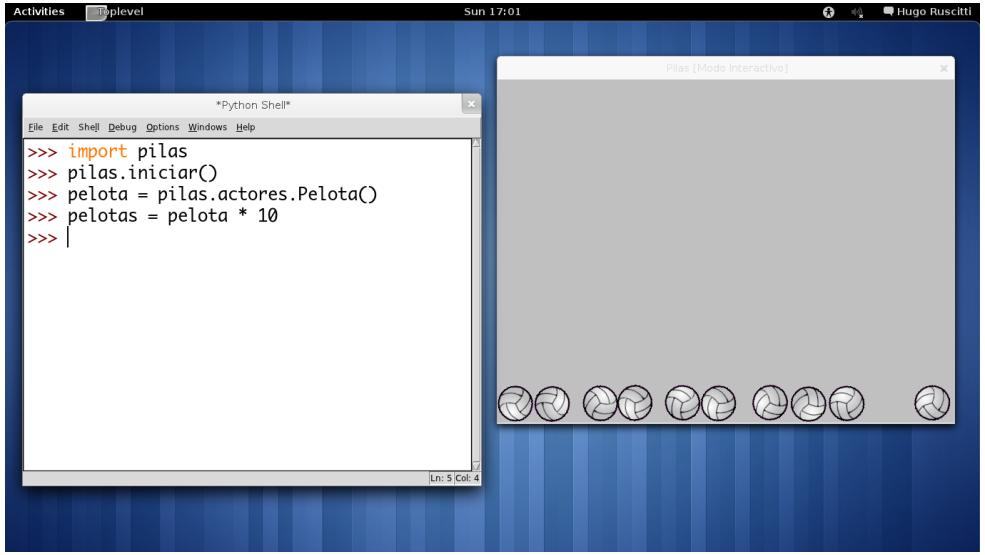
The function `iniciar` is responsible for opening a window, and offers some optional arguments if we want to specify them (this is not the case.)

Once we have the window, we could create an animated character to have some action:

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

What you see on screen is a bouncing volley ball alone on the floor. And if we want to bounce it against something else we could multiply it to have a lot bouncing balls:

```
pelotas = pelota * 10
```



And what about gravity?, you can change it easily as follows:

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

Where 10 and 30 are horizontal acceleration and vertical respectively. Indeed, the usual values are usually 0 and -90.

A more specific example

Consider this a little more in detail, write `pilas.reiniciar()` to clean up what we have in the screen.

Now, the actors are objects that live in the module `pilas.actores`. For example if we write:

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

You will see a little monkey in the middle of the window, and because we created it using a reference we will be able to indicate things such as the following:

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

We could also alter some visual properties like rotation, size, position, transparency etc. Take this example, let's move the monkey to the right of the screen and double its size:

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

Did you notice that the changes are immediate? How do we create animations?. You just have to change the integer numbers by lists:

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

The first sentence moves the monkey to the center of the window (horizontally), the second statement makes the monkey move up and then down. And the last sentence makes the monkey turns a full circle.

All actors are objects

The example above shows that the actors, are actually objects, have properties `scale`, `x` and `y`, but also have behavior, as the methods `decir` or the interpretation of messages as * (same as the numbers and strings) as we saw in the example of the volley ball.

This is a very powerful programming idea, because means that when you master one actor, in fact, you are learning to handle many actors, and in turn you are programming in python!

Researching

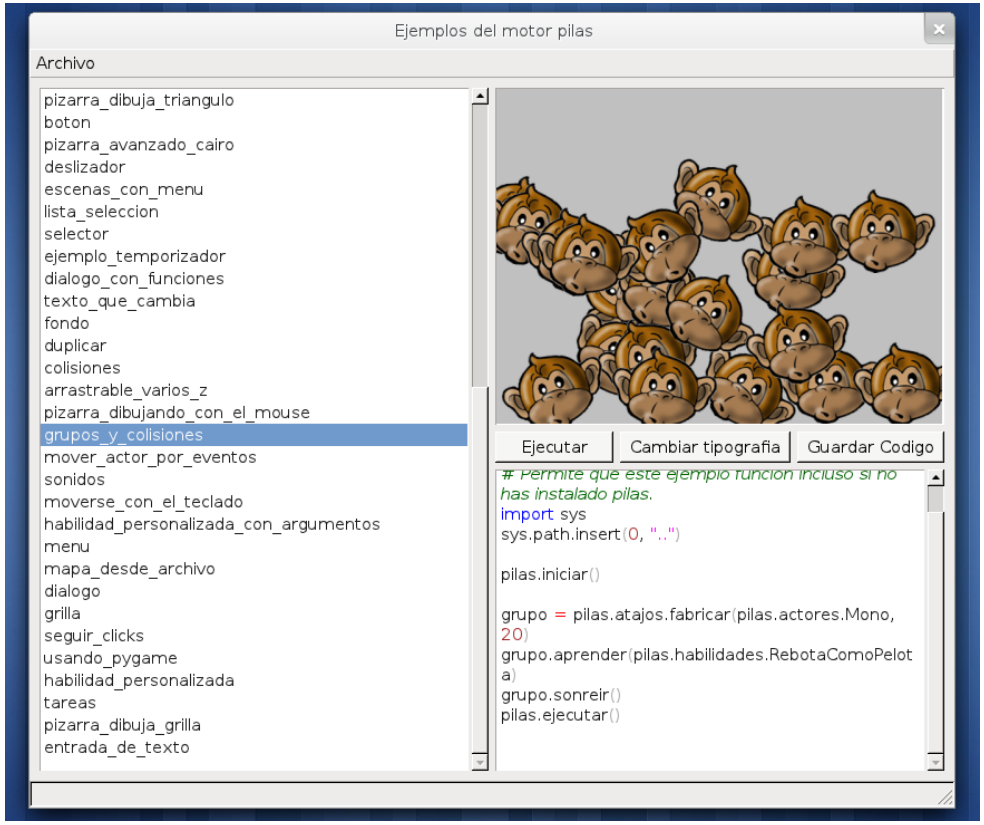
There is a function of `pilas` that comes useful when you start investigating: the function `pilas.ver`:

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

This feature can be used to inspect the source code of anything: modules, functions, actors, object references to classes etc ... How does it work?, easy, issue a `pilas.ver (pilas.ver)`.

And of course there is the function `help` and code auto-completion.

Even if you type `pilas` in a terminal, you'll notice an application that lets you see all the examples of code including pilas:



Looking ahead

I believe this is an interesting opportunity to show what fun and interesting programming is. A possibility of giving people a very powerful tool to move from consumers to producers of technology.

Just that at this point, it is not something purely technical, because the challenge is precisely disseminate, create and help inside of `pyar` to encourage people to participate.

For this reason, if you liked what you saw in this article, my advice is to encourage to write and tell more people about game development.

You could visit the website of pilas (www.pilasengine.com.ar), tell the pyar mailing list what you think or any ideas you have. Your comments are valuable for this and other projects can then move on.

pycamp ORM, making sqlalchemy easy to use

Autor: Emiliano Dalla Verde Marcozzi

Bio: The author started working with Python using Plone, being those also his first steps in programming.

Email: edvm@airtrack.com.ar

Twitter: @edvm

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

What's this article about?

Essentially three things:

- ORM / Object Relation Mapper, what is it?, what is it good for?
- SQLAlchemy, a Python ORM
- pycamp.orm, 140 lines (or so) that make SQLAlchemy easier to use

What are these three things for? To work with relational databases from different manufacturers such as:

- PostgreSQL
- MySQL
- Oracle
- SQLite

Our goal

Conquer the world? Yes, maybe, but not in this article ;). Our goal is to work with relational databases using Python and to have a general idea of the different actors who play a role in this scene, so we need to meet:

- Jenna Jameson, errhh I mean, SQL / Structured Query Language
- Database (PostgreSQL, MySQL, etc)
- Python's Connector / Driver to connect to the Database
- Python

The SQL / Structured Query Language is an ANSI (American National Standards Institute) standard which allows us to perform queries, define and modify data on a relational DB. Now, this is pretty neat!, we have a standard! This is: the databases which implement this standard should understand the commands that we program in this language.

Second: Databases which PARTIALLY implement SQL. This means that they may have some other words added to the SQL they implement. For example, MySQL may have sentences that don't exist in PostgreSQL.

The Connector / Driver (i.e. psycopg2, MySQLdb) is a piece of code which allows Python to communicate with the Database. The tasks that the connector handles are, for example, the work with sockets and many other operations which I don't really know anything about ;).

Python is the almighty programming language with which we could conquer the world, but in this article we'll use it to work with relational databases.

Now that we've met the actors, we could create a super cool ASCII diagram to represent how these actors relate:

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

To explain the first case of our super ASCII diagram, reading it from right to left: we have our almighty language Python which, using the MySQLdb connector, makes queries using sql_lang_1 to the MySQL database. Simple, right?

Same queries, different databases

Let's think of a hypothetical situation in which we need to create a program which should store the data it generates in a relational database.

It turns out, also, that the company where we work is using MySQL, so we're going to write our queries using the MySQLdb connector.

After 1337 months of development we have our program completely functional, but it turns out that our boss wakes up with a MySQL-incompatible mood and commands us to use PostgreSQL and that any software using a database other than PostgreSQL shall be sentenced to play on TRON's Arena!

Ok, so we and our little program are in a tight spot. Should we change all the queries we had written for MySQL to work with PostreSQL or run to get a frisbee (http://en.wikipedia.org/wiki/Flying_disc) and start practicing for our battle on TRON's

Arena (http://en.wikipedia.org/wiki/Tron_(film))?. Funny story aside, and to go back to my talk on PyDay Córdoba, we can say:

- Each database has its own dialect.
- In Python we use drivers/connectors to speak the dialect of the database with which we want to connect and use.
- Writing SQL code using a DB dialect doesn't seem to be a good idea, because when we need to change the DB engine we are forced to translate all our queries to the dialect of the new DB.

SQLAlchemy, the ORM used by MacGyver and Mr. T.

There's a solution for the problem of writing queries according to the dialect of our DB engine and it's called SQLAlchemy.

SQLAlchemy is an ORM, that is O*bject *R*elation *M*apper: a library which *translates the relationships between data in our DB to a set of objects. Let's see an example to make it clearer:

Let's say that the table "Person" in our DB has two columns:

- Name which is a `VARCHAR` field.
- Birthdate which is a `DATE` field.

That's for the relational aspect, now if we want to represent that data structure with objects in python we could have the following:

```
import datetime

class Person(object)
    """ I'm not only a cute face, I'm a person!
    """
    name = ''
    birthdate = datetime.datetime.now()
```

The job of the ORM is precisely to read the tables, columns, properties and relationships between data and express it using Python objects. This is very good for us, because as we will see, we're going to work with Python objects instead of writing SQL. It will be the ORM's responsibility, then, to translate everything to SQL to interact with the DB. Let's see this with a super enlightening ASCII graph:

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

So: we write Python and SQLAlchemy translates to the dialect of our DB. For example: a select on our Person table will be written as follows:

```
>>> from edo import Session
>>> from limbo import Person

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

In this sample we've imported our Person class and an invented Session with which we've performed our query. It should be noted that we worked with Python and not with our DB's dialect (that's SQLAlchemy's job ;)).

What we mean to say is that the python code we've written before, thanks to SQLAlchemy, works on PostgreSQL, MySQL or Oracle without having to change anything. Isn't that nice? ^^'

Doing Alchemy, ingredients

To start using SQLAlchemy we need to understand the following four elements: the four classical greek elements (earth, water, fire and air) go back to the pre-socratic times, live through the Middle Ages until the Renaissance having a deep influence in the European culture and thought. But that's not the case with SQLAlchemy, because here the four elements we need to understand are:

- Sessions
- Mapper
- MetaData
- Engine

Sessions: The sessions are meant to open and close connections to the DB. This is not accurately true, since SQLAlchemy keeps a pool of persistent connections to the DB, so that when we create a session we'll be taking one element of that pool and when we close the session we'll be giving it back. For that reason creating sessions is very fast and inexpensive, because the connections are already established and available in our pool.

Mapper: It's a state-of-the-art piece of code which maps the structures and properties on our DB to Python objects and viceversa.

MetaData: For the moment we will let this object in the dark.

Engine: It's an object where we define the properties of the connection to our database, such as: username, password, database name and database engine (Oracle, MySQL, PostgreSQL, etc.). We can also define the number of persistent connections in our pool and many other good things.

Order of the ingredients

Like in any other alchemistic procedure, the order is very important and that's why here we give you the order of the ingredients in SQLAlchemy:

1. Create the engine
2. Bind to our engine
3. Ready :)

The first thing we do is declare the engine. Of course, that's where we define the username, password, host, DB engine and all that jazz, so it makes sense that that is the first thing we do.

```
[1] from sqlalchemy import create_engine

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

On line [1] we import create_engine from sqlalchemy, on line [2] we create a url which defines the connection information like this:

```
url = 'DATABASEENGINE://USERNAME:PASSWD@IPHOST/DATABASENAME'
```

Lastly, on line [3] we call the `create_engine` method with our url as a parameter.

Let's see this with an example of real life code.

For that I will be using a virtualenv where I will install sqlalchemy (to learn more about virtualenvs see: 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...
```

Now I run ipython and we start to work:

```
(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)
```

As we can see, we have obtained the first item that we needed in our quest, now we have to go for the sessions, the mapper and the metadata. Let's go get 'em!:

```
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)
```

To explain the above code, we imported MetaData and saved in the meta variable the configuration of our MetaData with our engine. We've also imported sessionmaker and have passed to it our engine as a parameter. Remember that the first step was to create the engine and the second one to bind our metadata and sessions to the engine? Now we can create sessions calling our Session, which will take one of the persistent connections from the pool so that we can make queries. If on that session we execute the `close()` method we will return the connection to the pool. For example:

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

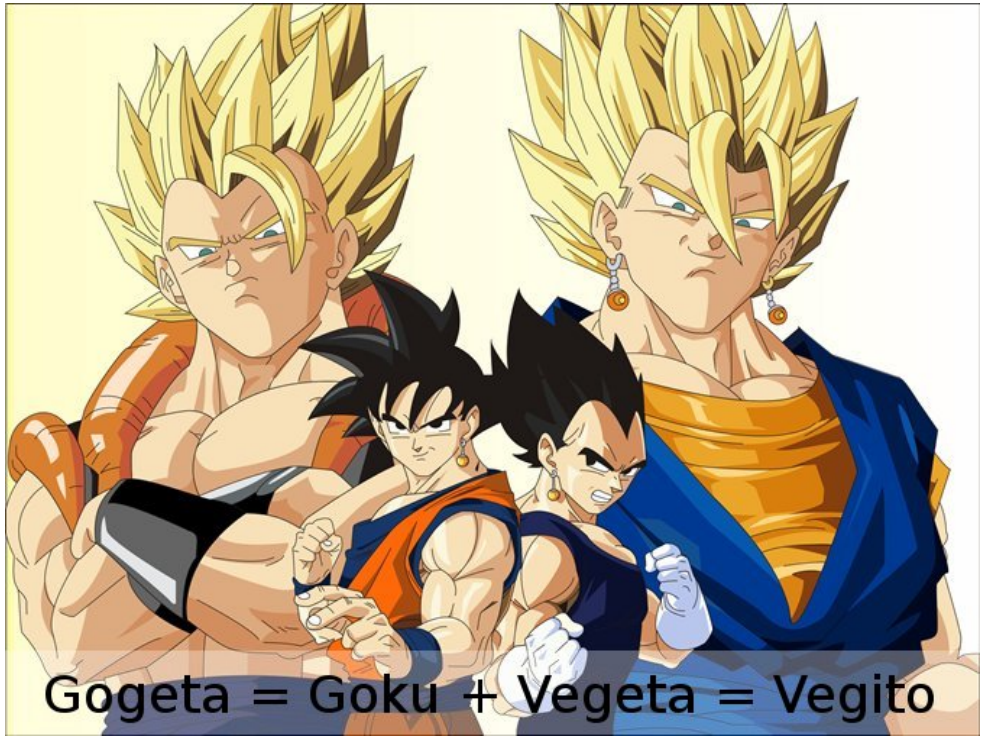
We're still missing the mapper, so let's take a look to what the docstring says about it:

```
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.**
```

The mapper's documentation tells us that it takes two parameters, the first one being a class and the second one a local_table. For those of you who watched Dragon Ball, understanding this will be very simple... mapper is like the fussion between Goku and Vegeta to form Vegito (http://dragonball.wikia.com/wiki/Vegito) or Gogeta (http://dragonball.wikia.com/wiki/Gogeta). You put one Goku, one Vegeta and you get one Vegito/Gogeta. Easy, right?



Ok, for those of you who didn't watch Dragon Ball, this would be like making an orange juice: you put a juice extractor, some oranges and you get an orange juice. You see how easy was this mapper thing? ;)

To continue with the explanation, local_tables are the tables on our DB and the class is a custom class that will be binded to the table in our DB, and what we'll get as a result of passing to mapper our class and our local_table is: a `Mapper` object. Let's try to see this with an example:

```
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'>
```

In this example we have connected to a database of a Django project, and let's pay attention to lines [5] and [7].

On line [5] we defined our metadata passing as parameters our engine and `reflect=True` which makes SQLAlchemy become a Super Saiayin (NOT an orange juice) and connect to our database magically discovering all of our tables and adding them to a dictionary inside meta.tables where all the keys are the tables' names and the values are objects of the type sqlalchemy.schema.Table.

Now we have the local_tables of our database and what we need is to create a class for every table, so that we can have the two parameters needed for the mapper. So let's get into the loop:

```
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]) # we pass the class and the 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'>
```

In the previous lines we created a DB class which will be used to store the mapped tables of our database. On [13] we instantiate DB, then on [17] we import mapper and on [18] we loop through every element of the dictionary meta.tables. This is, we're looping through every table that was auto-magically discovered by MetaData's reflect, creating on the fly a new type using the type method, assigning to it the name of the table. Then, using setattr we put the recently created object in our instance db and, finally, we call mapper passing to it the object we created with type as the first parameter and the object of type sqlalchemy.schema.Table (that is, our local_table) as the second parameter.

To finish this part we can see on [19] as an attribute of db our already mapped table auth_user.

And that's not it, we could also do a query like this:

```
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()
```

What about pycamp.orm?

Well, pycamp.orm are about 140 lines that at the time of writing this article I realized that they could be even less.

It does exaclty what it's shown in this article (hehehe). So you already know how pycamp.orm works ;)

This module was born on the PyCamp that was hosted in La Falda, Córdoba, Argentina; which was a camp that gathered Python programmers.

It was my first PyCamp and I had a great time, the people were really cool. Besides it's really good meeting in person with people that you chat with on IRC, meeting new people, playing role-playing games and being constantly treated as either a mutant or a communist (on the role-playing game, I mean ;)), losing terribly on foosball championships, learning how to juggle and, yes!, coding stuff that YOU LIKE!. So, let's see how to use the current version of pycamp.orm:

```
(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()
```

Let's explain a little what pycamp.orm does: From mapper we import a `Database` which is where we set the username, password, database engine, etc. All the data that we store on Database is what will be used with the method create_engine of SQLAlchemy ;). In [4] we create a `DatabaseManager` which is a BORG (http://code.activestate.com/recipes/66531-singleton-we-dont-need-no-stinkin-singleton-the-bo/) that has an `add()"method that takes a "Database` as a parameter from which it takes the information to create the metadata; it calls the sessionmaker, the mapper and leaves everything ready to start querying the DB :).

Well, that's it. To finish this article:

The code is here:

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

You can see the video of this article which is a little different (and in Spanish) here:

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

You can download the slides of the talk here (also in Spanish):

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

And that's it, I hope you liked it!

VIM, an editor with batteries included

Author: Hugo Ruscitti

Bio: Informatics Student in the Universidad Tecnológica Nacional (UTN: National Technology Univ.). He has worked creating linux distributions and is currently an associate in gcoop work cooperative. His greatest hobby is videogame programming, activity he promotes and carries out together with his site's members at http://www.losersjuegos.com.ar/

Email: hugoruscitti@gmail.com

Web: www.pilas-engine.com.ar

To write python code you could use any text editor, there are a lot of great options and different approaches to development

But from all the editors I know, there is one that deserves special attention, not only because of its huge user-base but also because of its features and quirks. Yes, I'm talking about VIM.

Why VIM?

If you use a computer a lot, and spend most of that time programming, you will surely care about the editor you use. The more experience you have with writing code, the less you want to repeat yourself and want to focus on making important changes as fast as possible.

VIM is a text editor developed for those cases, an editor written by programmers (obviously) and for programmers. One of the keys to its success are its commands and plugins to make day-to-day tasks quicker with as little key-stroking and as directly as possible.

For example, in Python's case, it makes browsing the structure of very big projects really simple, has code auto-completion and eases making changes of portions of the code.

A simple example

Picture this: you downloaded a python library from a repository and are investigating how it works.

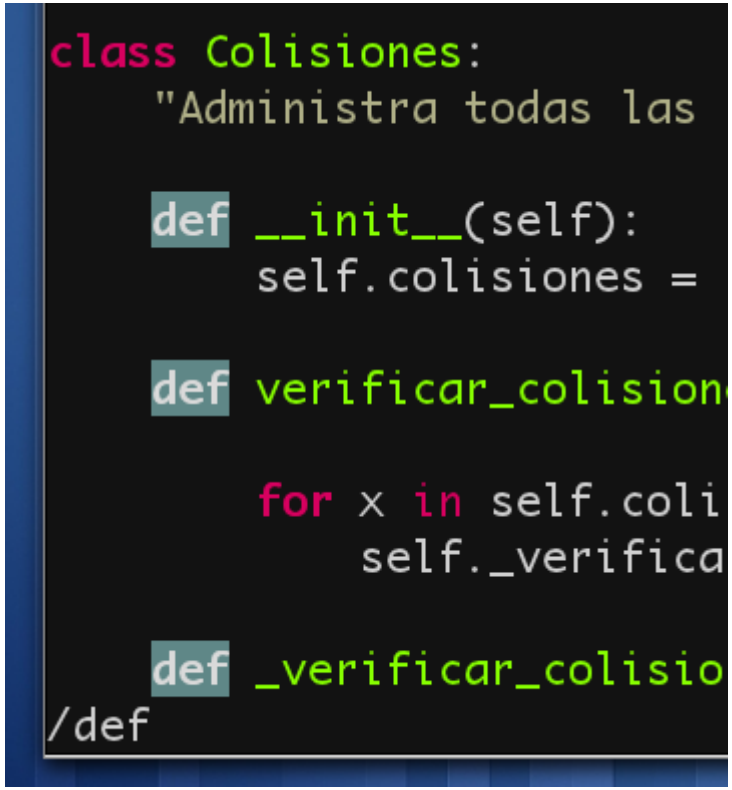
To open VIM and explore the project's files you could write something like the following in your console:

```
vim projects_folder
```

And the editor will show a list of files to inspect. If you press ENTER on any file it would just open it. And it does syntax highlighting.

If you now want to go to a specific portion of the code, it is very likely that you have an idea of the text you expect to fine. For those cases, VIM has quick-searching that you activate pressing "/".

For example, if you want to browse through the functions of a file you write "/def" (the "/" starts searching and "def" is what identifies python's functions).



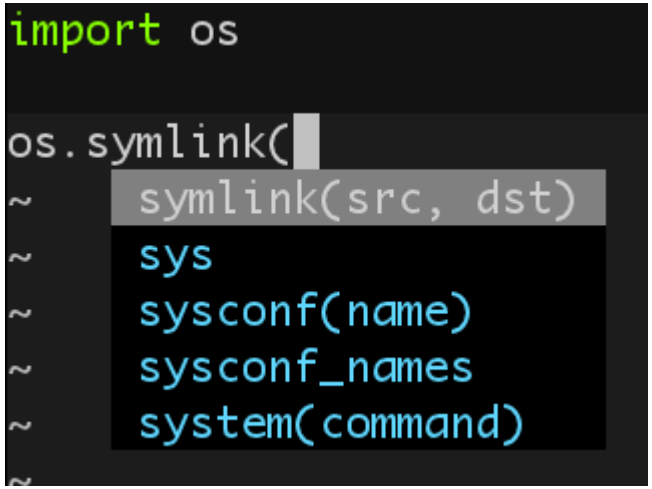
Now, pressing ENTER will stop searching. If you press “n” it will search for the next match, and “N” the previous one.

Lets imagine that there is a method name you are interested in. If you press “W”, VIM will jump to the next word (the name of the method or function) and then pressing “*” it will jump to the next appearance of that function.

This last shortcut is very usefule to highlight reference names in a portion of the code.

Auto-completion

Auto-completion is quite complex, you can auto-complete filenames using `CTRL+x CTRL+F`, complete words by repetition with `CTRL+p` or even use context-based auto-completion, for example within a python program pressing `CTRL+x CTRL+o`:



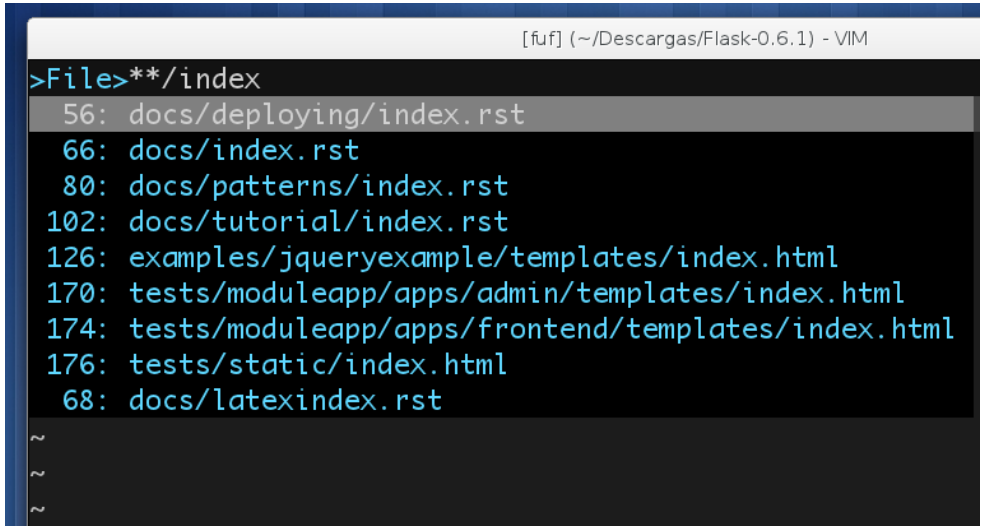
The only bad thing about these auto-completion types, is that they are a tad complicated to invoke. A way to overcome that issue is to install a plugin called SuperTab (http://www.vim.org/scripts/script.php?script_id=1643) that invokes auto-completion when you press the TAB key (and is intelligent enough to allow us to keep using TAB for indenting).

Quickly locating files

Something quite common in software projects is that you need to access and switch between several files.

VIM has several options for that: you can use tabs with commands like `:tabnew file_name` or split your window with commands like `:split` or `:vsplit`.

But if you want something immediately, a plugin like FuzzyFinder (http://www.vim.org/scripts/script.php?script_id=1984) comes handy as it allows you to find files quickly while you write its name (no matter the folder):



There is even another plugin called MRU (http://www.vim.org/scripts/script.php?script_id=521) that opens a window with recently-modified files.

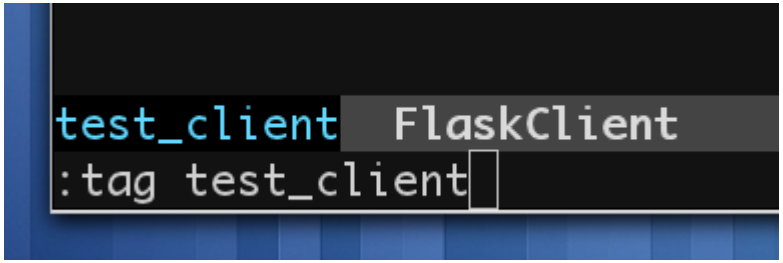
Quick navigation

VIM allows for navigation within a big project jumping from a function's invocations and its declaration.

To use that feature, we first need an external tool called `ctags` that prepares the project to be explored:

```
cd projects_folder · ctags -R *
```

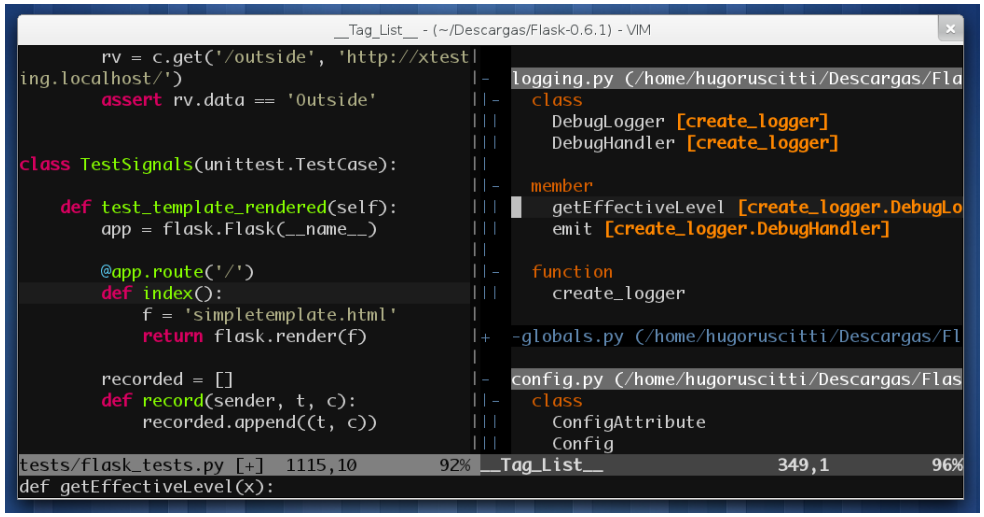
This enables a lot of features; for example, we could open the project with the `vim .` command and then write `:tag *cli` and press TAB. VIM will locate all classes, functions or references that have `cli` in them. En my case, it is offering two symbols; if I pressed ENTER it would jump directly to their declarations:



Another interesting functionality that is enabled by the use of ctags is that you can place your cursor over the name of any class, function or reference and pressing `CTRL+]` will make VIM jump directly to where it is defined. Just like a link!

Possibilities just cascade from here, because now the editor knows exactly where each thing in your program is defined and can take advantage of it.

You can even install something like TagList to see a panel with your class' interfaces:



The right portion of the previous image shows a tree that provides direct access to several classes. That is very useful when you are working with several big classes and you need to jump all around following your application's flow.

Configuring your environment

It is important to mention at this point that one of the main points of VIM is its configuration.

Unfortunately, the editor has a very bad default configuration so it is normally a very good idea spend a few minutes researching about configuration parameters in blogs that belong to python programmers that use VIM.

For example, the following option in your `~/.vimrc` file means that pressing `<F2>` will make the script you are editing at the moment to be run, something very useful when prototyping:

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

Here are some links where people suggest how to customize the 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

Conclusion: be patient

VIM has a lot of interesting features and it would take a life to see all of them. It takes some time since you start using it until you feel comfortable with it, but is becomes more and more valuable and fun with time. It makes you think about quicker and simpler ways to do stuff.

So, if you are searching for a good text editor, my recommendation is that you take VIM into account and start using it with some patience...

What does it feel like to be a speaker?

Author: Tomas Zulberti
Bio: I'm Tomas Zulberti, alias tzulberti. There's not much to say more than I'm student at UBA in Computer Sciences, and nothing...
Email: tzulberti@gmail.com
Twitter: @tzulberti
Webpage: http://tzulberti.com.ar/

For those who do not know me, my name is Tomas Zulberti and I was speaker at PyDay of Cordoba in 2011 with the same talk as in PyCon 2010: Using additional libraries.

As there was an article about my talk at the PET #2; I thought I could discuss my experience as speaker (Which is not much compared with that of others).

The first talk I gave was in Rafaela PyDay 2010 (http://www.pyday.com.ar/rafaela2010). Unlike the events in Cordoba, the talks were 40 minutes and had only one single track of talks. It was a very really bad presentation for several reasons:

- Talk blended many different topics:
    - Versioning Systems: svn, and mercurial
    - The Python IDEs: vim, pycharm and eclipse (Ninja did not exist).
    - Different consoles of python: ipython, bpython, etc.
    - How to install libraries: easy-install and pip
- I was very nervous. Although I had practiced the conversation at my home is not the same when one has come forward to speak in front of 70 people.
- As I had too many spots, I could not speak a lot about each.

When finished and questions from the audience came, I realized the talk had not been good. But I learned an important lesson:

It's easier if you focus on few topics.

This allows you to give more details of how things work. Otherwise, you just get to name features that may be relevant for your audience.

My second talk, shared with Facundo Batista, was in the event organized in radio La Tribu (http://python.org.ar/pyar/CharlasAbiertas2010) which had the title "Introduction to Python" and agreed by mail that I would show some things from the python standard library.

I investigated the documentation on modules with "simple" functionality to explain and chose the following:

- datetime: To work with dates.
- mail: Everyone knows what an email is and it seemed a good idea to show how with python one could use the service in less than 15 lines.
- zip: To compress files.
- unittest: For unit testing.

I also gave a list of some other useful modules.

And how I was this time? I was nervous again, but I liked the topics I gave. I also talked very fast (I can not remember if it was because I thought I had no time or because nerves).

As we head into what it was my third talk which was in PyDay Buenos Aires 2010 (http://www.pyday.com.ar/buenosaires2010), I decided to give the same topics proposed for PyCon Cordoba (external libraries) and use this event as a practice. I wasn't that nervous here as in previous events, but the talk was not well given. I complicated my talk a bit because one of the examples I had to give did not work, though at 18 minutes I had finished with the questions of a total of 25 minutes I had assigned.

Finally, the event which was my goal came: PyCon 2010. The talk, as I mentioned, was the same about external libraries. Was the turn to expose on Saturday and unlike of previous events I liked how I gave it.. I had prepared everything for the examples not to fail and I went well with timing. I have to admit that I not really liked how I responded to more than one question.

Finally, I gave the same talk at the PyDay Córdoba 2011. This time the talk was good (from my point of view) and by the questions asked I was able to see that people did understand what I spoke.

Wrapping up

Some important points to encourage you to give talks are:

- In the event we are organizing the talks do not have to take more than 25 minutes.
- You do not have to know everything about what you're talking about. There is no problem if someone asks a question and you respond "I don't know".

- Must choose a topic that you commonly use.

Take heart to give a talk. Currently there is a group of mentors (http://python.org.ar/pyar/AdoptaUnNewbie) that can help you prepare one.