Nano Taller de Python

Charla 1: “Introducción a Python”

Sergio Davis <sergdavis@gmail.com>

Royal Institute of Technology (KTH), Estocolmo, Suecia
Grupo de Nanomateriales (GNM), Santiago, Chile

12 de enero 2009, de 10:00 a 11:00

Programa para el Taller

El programa para hoy es el siguiente:

10:00 - 10:50	Charla 1: Introducción a Python
10:50 - 11:00	*coffee break*
11:00 - 12:00	Práctica 1: Familiarizarse con el ambiente Python
12:00 - 13:00	*Almuerzo*
13:00 - 13:50	Charla 2: Programación estructurada
13:50 - 14:00	*coffee break*
14:00 - 14:50	Práctica 2: Condicionales, ciclos y funciones
14:50 - 15:00	*coffee break*
15:00 - 15:50	Charla 3: Listas, tuplas, diccionarios
15:50 - 16:00	*coffee break*
16:00 - 18:00	Práctica 3: Listas y diccionarios, *parseo* de archivos

Esquema de este Taller

El programa para mañana es el siguiente:

10:00 - 10:50	Charla 4: Programación Orientada a Objetos
10:50 - 11:00	*coffee break*
11:00 - 12:00	Práctica 4: Clases, herencia, *duck-typing*
12:00 - 13:00	*Almuerzo*
13:00 - 13:50	Charla 5: Programación funcional
13:50 - 14:00	*coffee break*
14:00 - 14:50	Práctica 5: Comprensiones de listas, generadores
14:50 - 15:00	*coffee break*
15:00 - 18:00	Práctica 6: Proyectos

Elementos que “asumo” para este taller

(Eso sí, con bastante flexibilidad)

- Programación en algún lenguaje, ya sea...
  - De tipo estructurado: BASIC, Pascal, Fortran o C
  - Orientado a objetos: C++, Java
  - De scripts: bash, awk, Perl
- Ningún conocimiento previo de Python
- Alguna experiencia con UNIX / Linux, básicamente saber editar y ejecutar programas
- Manejo de `vi` o `emacs` siempre ayuda pero no es estrictamente necesario

Nota:

En las demostraciones voy a usar `vim` y `gvim`, pero el editor es a comodidad del usuario. Para los indecisos, sugiero `eric4`, un editor para Python escrito en Python.

Parte I

Introducción a Python

¿Qué es Python?

- Python es un lenguaje de programación dinámico creado en Holanda en 1991 por Guido van Rossum.
- Soporta tres estilos o paradigmas de programación: *estructurada*, *orientada a objetos*, *funcional*.
- Usado en todo tipo de aplicaciones: científicas, administración de sistemas, procesamiento de texto, páginas web, bases de datos, visualización 3D y videojuegos, inteligencia artificial, etc.
- Presenta una sintaxis compacta, sencilla e intuitiva, una curva de aprendizaje mínima, junto a una potente librería de funciones y clases.
- Lo anterior hace posible programar una aplicación completa en cosa de horas o incluso minutos.

Popularidad de Python

Position Nov 2008	Position Nov 2007	Delta in Position	Programming Language	Ratings Nov 2008	Delta Nov 2007	Status
1	1	=	Java	20.299%	-0.24%	A
2	2	=	C	15.276%	+1.31%	A
3	4	↑	C++	10.357%	+1.61%	A
4	3	↓	(Visual) Basic	9.270%	-0.96%	A
5	5	=	PHP	8.940%	+0.25%	A
6	7	↑	Python	5.140%	+0.91%	A
7	8	↑	C#	4.026%	+0.11%	A
8	11	↑↑↑	Delphi	4.006%	+1.55%	A
9	6	↓↓↓	Perl	3.876%	-0.86%	A
10	10	=	JavaScript	2.925%	0.00%	A
11	9	↓↓	Ruby	2.870%	-0.21%	A
12	12	=	D	1.442%	-0.26%	A
13	13	=	PL/SQL	0.939%	-0.24%	A
14	14	=	SAS	0.729%	-0.40%	A--
15	18	↑↑↑	ABAP	0.570%	-0.08%	B
16	19	↑↑↑	Pascal	0.511%	-0.13%	B
17	17	=	COBOL	0.510%	-0.20%	B
18	25	↑↑↑↑↑↑↑	ActionScript	0.506%	+0.04%	B
19	23	↑↑↑↑	Logo	0.489%	-0.04%	B
20	16	↓↓↓↓	Lua	0.473%	-0.27%	B

Ramas de desarrollo de Python

Python 1 Rama obsoleta (última versión lanzada en abril 1999), no hay razón para usarla.

Python 2 Es la rama que todo el mundo usa. Versiones comunes son 2.3, 2.4 y 2.5, compatibles entre sí excepto pequeños detalles.

Python 3 Lanzada el 4 de diciembre 2008, diseñada a propósito para romper compatibilidad con Python 2, muy poca gente lo usa. Próxima versión estándar una vez Python 2 sea declarado obsoleto (no en un futuro próximo . . .)

Nota:

Nosotros trabajaremos con Python 2.5, un buen compromiso entre la compatibilidad con librerías existentes y las características nuevas del lenguaje. Prácticamente todo lo que veamos también funciona en Python 2.4 y 3.0, salvo pequeños detalles de sintaxis.

¿Por qué usar Python?

- Diseño simple e internamente consistente
- Código usualmente muy claro y legible
- Alto nivel: operaciones complejas en muy pocas líneas
- Módulos que facilitan un espectro de tareas
- Versatilidad: distintos paradigmas de programación

Además...

¡Hoy en día todo el mundo usa Python! Debian 5.0 tiene más de 750 módulos no oficiales para Python, además de los incluídos en la librería estándar.

Simplicidad como filosofía base

Extraído del “Zen de Python” . . .

- Hermoso es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- La legibilidad importa.
- Los casos especiales no son tan especiales como para romper las reglas.
- Aunque lo práctico le gana a lo purista.
- Debería haber una– y de preferencia sólo una –manera obvia de conseguir algo.

(la versión completa en inglés aparece con `import this`)

Python versus otros lenguajes

```
program selectrandom

real suma, x
suma = 0.0
call srand(0)
do i=1,500
   x = 2.0*rand()-1.0
   if (x < 0.0) then
      suma = suma + x
   end if
end do
write (*,*) suma

end
```

Suma de números aleatorios negativos, en Fortran 90

Python versus otros lenguajes

```
#include <iostream>
#include <cstdlib>

int main()
{
 srand48(long(time(NULL)));
 float s = 0.0;
 for (int i=0;i<500;++i)
 {
  float x = 2.0*drand48()-1.0;
  if (x < 0.0) s += x;
 }
 std::cout << s << std::endl;
 return 0;
}
```

Suma de números aleatorios negativos, en C++

Python versus otros lenguajes

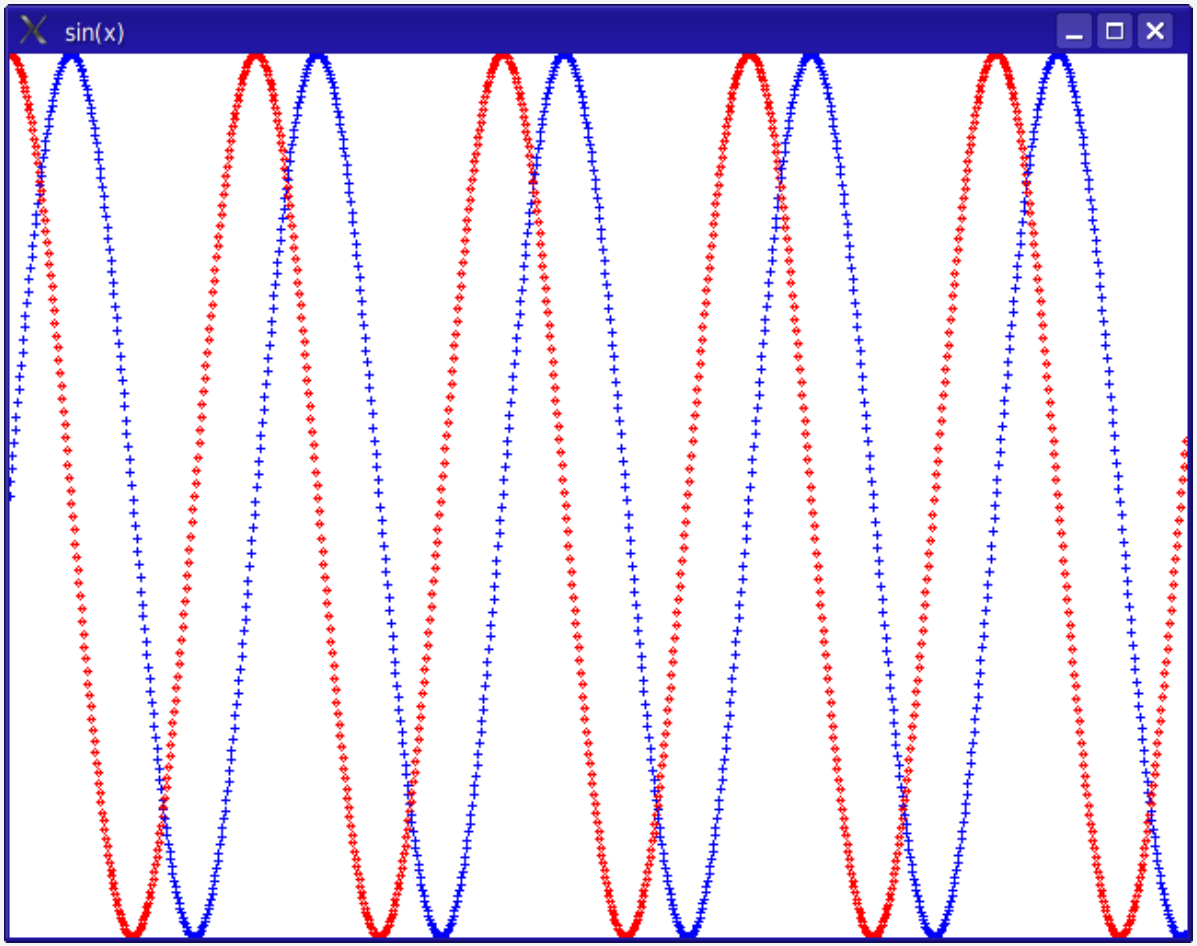
```
from random import random

s = 0.0
for i in range(500):
    x = 2.0*random()-1.0
    s += (x if x < 0.0 else 0.0)
print s
```

Mismo programa en Python (versión estructurada)

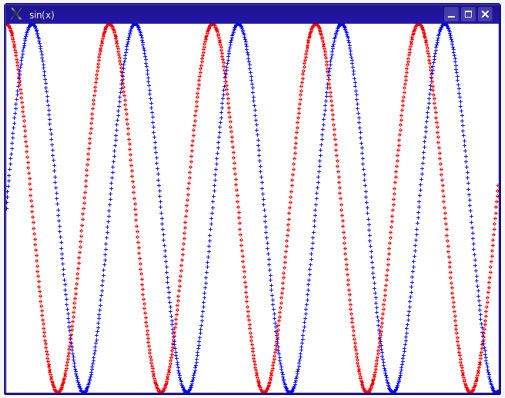
```
from random import random

num = (2.0*random()-1.0 for i in range(500))
print sum(x for x in num if x < 0.0)
```

Mismo programa en Python (versión funcional)

Ejemplo 1: `simpledraw`



Ejemplo 1: `simpledraw` (código)

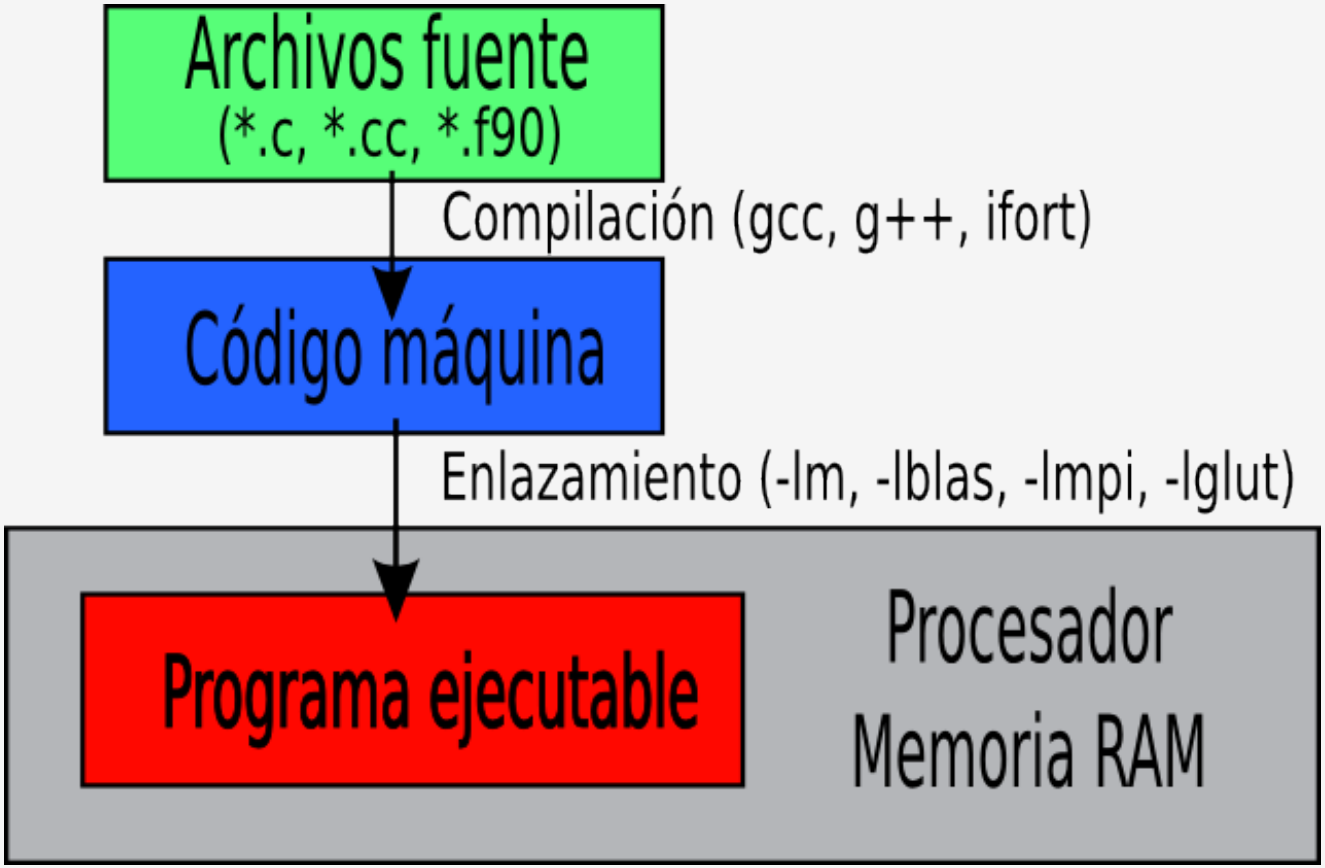



```
from math import sin, cos
from simpledraw import *
init("sin(x)")
setbackground(white)
for i in range(1000):
    x = float(i)/1000.0
    usecolor(blue)
    cross(x-0.5, 0.5*sin(30.0*x))
    usecolor(red)
    diamond(x-0.5, 0.5*cos(30.0*x))
finish()
```

Python es un lenguaje interpretado

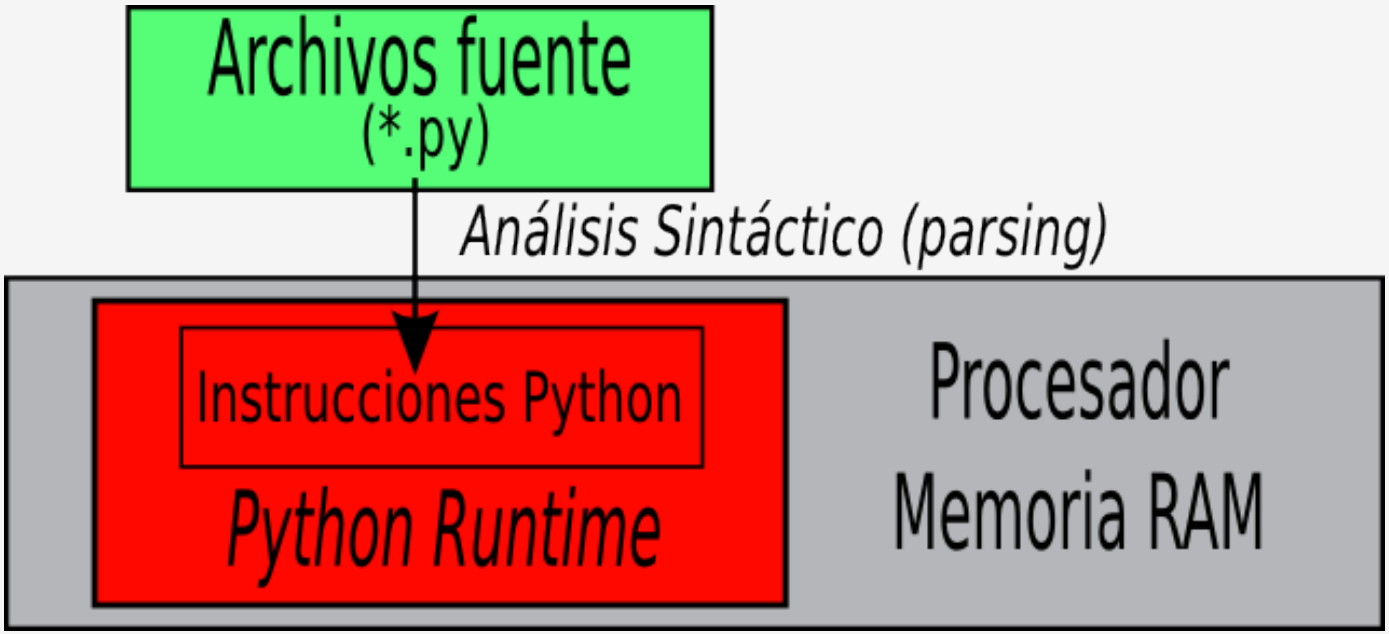
- Un programa Python no se compila, simplemente se ejecuta
- Esto hace posible cosas prácticamente imposibles en otros lenguajes:
  - Ejecutar instrucciones de Python interactivamente.
  - Interpretar un *string* como código Python y ejecutarlo.
  - Crear funciones y clases “al vuelo”, es decir, mientras el programa corre.
  - Introspección: un programa Python puede “analizarse a sí mismo”

¿Cómo funciona un lenguaje compilado?



- Al compilarse, el código fuente es traducido a una secuencia de instrucciones básicas que el procesador entiende (programa ejecutable), además de llamadas a librerías externas.
- Nuestro programa “vive” directamente dentro de la CPU.

¿Cómo funciona un lenguaje interpretado?



- El código fuente Python es analizado como instrucciones básicas para un "procesador virtual", el *Python Runtime*, que a su vez es un programa ejecutable.
- Es el *Python Runtime* el que "vive" directamente dentro de la CPU, y escondido dentro de él, nuestro programa.

Python es un lenguaje con tipos de datos “dinámicos”

En los lenguajes compilados como Fortran, C y C++ es necesario declarar de antemano las variables con su tipo de datos:

```
implicit none
integer n       ! entero
real x          ! real
```

Inicialización de variables, Fortran

```
int n;          /* entero */
float x;        /* real (precision simple) */
float * y;      /* puntero a un real */
```

Inicialización de variables, C

```
bool flag;      // valor booleano (verdadero/falso)
MiClase * m;    // puntero a un objeto MiClase
const std::string & texto; // referencia a un string
```

Inicialización de variables, C++

En Python las variables no llevan tipos

En Python no hace falta *declarar* las variables, sólo hay que *inicializarlas* con algún valor antes de usarlas:

```
# Python
n = 5
x = 137.0
texto = "hola mundo"
```

Inicialización de variables

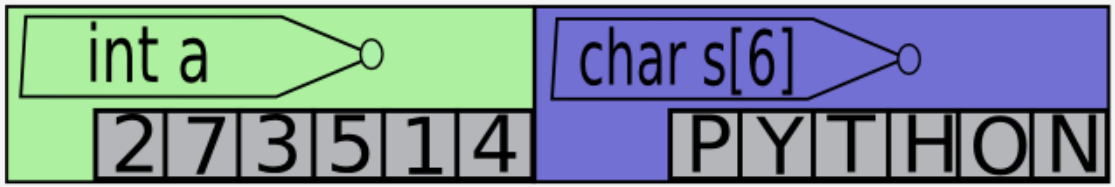
¡Esto no significa que Python no tenga tipos o que los ignore!

```
print x              # x no existe aun
x = 42               # x "apunta" a un entero
print x + 8          # imprime 50
print "hola" + x     # tipos incompatibles
x = "mundo"          # x "apunta" a un string
print "hola" + x     # imprime "holamundo"
```

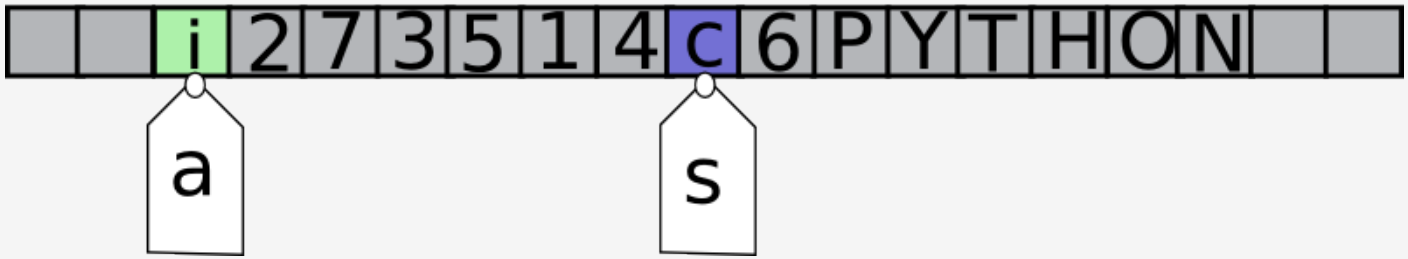
Ejemplo: Tipos dinámicos

En Python las variables no llevan tipos

Variables en lenguajes estáticos



Variables en lenguajes dinámicos



Es el objeto mismo en memoria y no la variable el que lleva “pegada la etiqueta” de su tipo.
En Python una variable es más como una *referencia* a un objeto, que puede apuntar a objetos de distinto tipo.

Python tiene recolección automática de basura

En los lenguajes compilados “modernos” es posible crear y destruir objetos en memoria de forma dinámica:

```
real, dimension(:), allocatable :: A
allocate(A(100))
...
deallocate(A)
```

Memoria dinámica en Fortran 90

```
float * A = (float *)malloc(100*sizeof(float));
...
free(A);
```

Memoria dinámica en C

```
float * A = new float[100];
...
delete [] A;
```

Memoria dinámica en C++

Manejo de memoria en Python

Memoria en Python

En Python todos los objetos se crean en forma dinámica y no es necesario liberar la memoria explícitamente (aunque es posible hacerlo).

Cuando un objeto deja de ser accesible (es decir, ya no existen variables que apunten a él) es liberado de la memoria automáticamente por el *recolector de basura* (*garbage collector*).

```
# A es una lista de 100 floats
A = [0.0 for i in range(100)]
...
# No es necesario liberar la memoria usada por A
# Si se quiere liberar manualmente, se usa:
#     del A
```

"Recolección de basura" en Python

Manejo de memoria en Python

La mayoría de las veces la manera más comoda es usar un contenedor, llenándolo a medida que se van generando los valores:

```
A = []                  # lista vacia
A.append(1.0)           # A = [ 1.0 ]
A.append(3.0)           # A = [ 1.0, 3.0 ]
A.append(5.0)           # A = [ 1.0, 3.0, 5.0 ]
...
# La lista va acomodandose a la cantidad
# de elementos que contiene
```

Memoria dinámica en Python

La gran ventaja de que Python maneje la memoria...

- No más fugas de memoria (*memory leaks*)
- No más violaciones de segmento (*segmentation faults*)

Bloques de código en Python

En otros lenguajes existen palabras clave o caracteres que marcan el principio y fin de un bloque. Por ejemplo,

```
do i=1,10
   if (i > 5) then
      write(*,*) i
   end if
end do
```

Bloques en Fortran

```
for (int i=1;i<=10;i++)
{
 if (i > 5)
 {
  std::cout << i << std::endl;
 }
}
```

Bloques en C++

La indentación en Python

En Python, la cantidad de espacio en blanco al inicio de la línea (llamado el nivel de *indentación*) es lo único que dicta el nivel de profundidad.

```
for i in range(1,11):
    if i > 5:
        print i
```

Bloques en Python

Indentación

No importa exactamente cuántos caracteres, sólo si aumenta o disminuye respecto a la línea anterior. Tampoco es necesario alinear el programa principal a la primera columna.

La indentación en Python

La decisión de hacer que el espacio en blanco al inicio de las líneas sea importante ha sido y aún es controversial...

Y lamentablemente ha alejado a mucha gente de Python (sobre todo a los programadores desordenados)...

Lo que no se debe hacer!

(en C, C++, Java, Perl, ...)

```
} else if (dummy_func) {
    if (equals(c_token, c_dummy_var[0])) {
    c_token++;
    add_action(PUSHD1)->udf_arg = dummy_func;
    } else if (equals(c_token, c_dummy_var[1])) {
    c_token++;
    add_action(PUSHD2)->udf_arg = dummy_func;
    } else {
    int i, param = 0;

    for (i = 2; i < MAX_NUM_VAR; i++) {
        if (equals(c_token, c_dummy_var[i])) {
        struct value num_params;
        num_params.type = INTGR;
        num_params.v.int_val = i;
        param = 1;
        c_token++;
        add_action(PUSHC)->v_arg = num_params;
        add_action(PUSHD)->udf_arg = dummy_func;
        break;
        }
    }
    if (!param) {   /* defined variable */
        add_action(PUSH)->udv_arg = add_udv(c_token);
        c_token++;
    }
    }
    /* its a variable, with no dummies active - div */
} else {
    add_action(PUSH)->udv_arg = add_udv(c_token);
```

Con llaves hay diferentes estilos...

```
int main(int argc, char *argv[]) {
    while (x == y) {
        something();
        somethingelse();
        if (some_error) {
            do_correct();
        }
        else {
            continue_as_usual();
        }
    }
    finalthing();
}
```

C al estilo Kernighan & Ritchie

Con llaves hay diferentes estilos...

```
int main(int argc, char *argv[])
{
    while (x == y)
      {
          something ();
          somethingelse ();
          if (some_error)
            {
                do_correct ();
            }
          else
            {
                continue_as_usual ();
            }
      }
    finalthing();
}
```

C al estilo GNU

Con llaves hay diferentes estilos...

```
int main(int argc, char *argv[])
{
 while (x == y)
 {
  something();
  somethingelse();
  if (some_error)
  {
   do_correct();
  }
  else
  {
   continue_as_usual();
  }
  finalthing();
 }
}
```

C++ al estilo LPMD

¡Python tiene un único estilo!

(o un estilo único...)

```
def main(argc, argv):
    while x == y:
        something()
        somethingelse()
        if some_error:
            do_correct()
        else:
            continue_as_usual()
    finalthing()
```

Python

- Hace el código naturalmente más legible y limpio
- Obliga a formatear correctamente un programa sin ambiguedades
- Un programador ordenado de todas maneras agrega espacios para hacer el código más legible

La sintaxis de Python es compacta

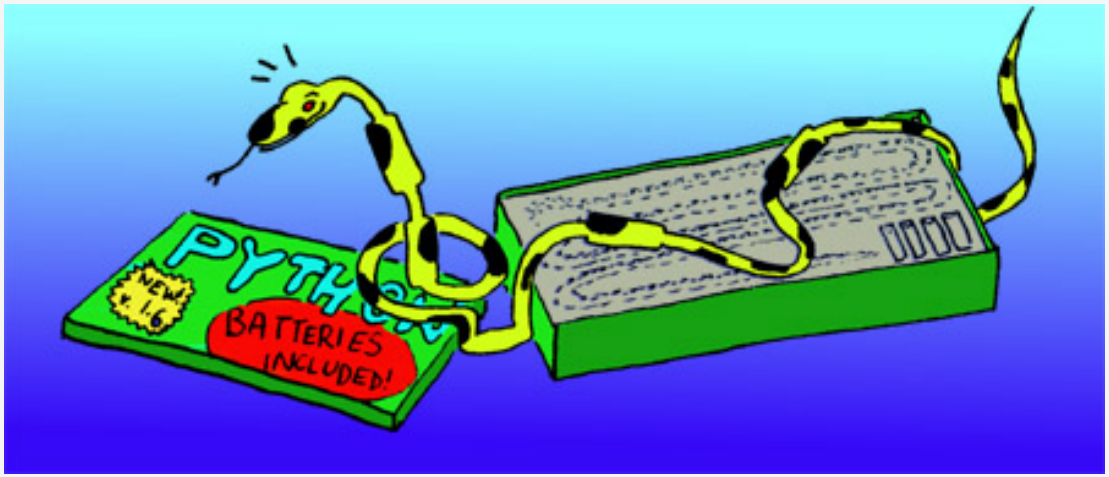
Toda la sintaxis de Python 2.5 se compone de 29 palabras reservadas:

```
and       elif      global    or        yield
assert    else      if        pass
break     except    import    print
class     exec      in        raise
continue  finally   is        return
def       for       lambda    try
del       from      not       while
```

Compare por ejemplo con C, que tiene 31, C++ con 47, Fortran y Java con 50.

Los tipos de datos básicos en Python son: `int`, `long`, `float`, `complex`, `bool`, `str`, `file`, `list`, `set`, `tuple`, `dict`.

Python viene “con las pilas incluídas”

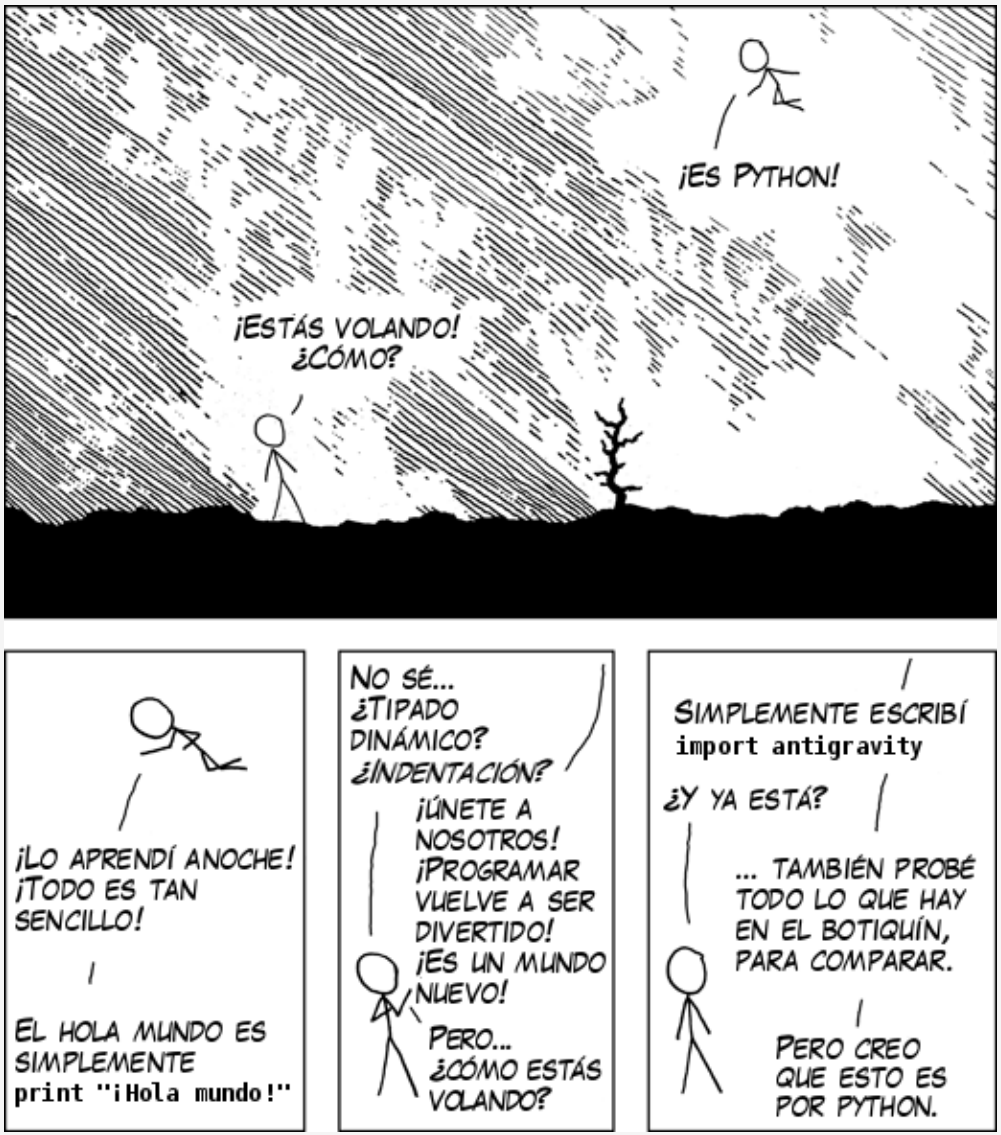


La *librería estándar* de Python incluye módulos para:

- Funciones matemáticas reales y complejas, números aleatorios, criptografía
- Manejo de conexiones de red (TCP/IP, Web, FTP, correo, etc)
- Compresión y descompresión de archivos (zlib, gzip, bzip2, tar)
- Manipulación de texto y expresiones regulares
- Acceso a bases de datos
- Acceso al sistema operativo, creación de subprocesos, manejo de archivos y directorios

Con Python, programar vuelve a ser divertido!

(XKCD comic número 353, “Python”)



Parte II

¿Cómo trabajar con Python?

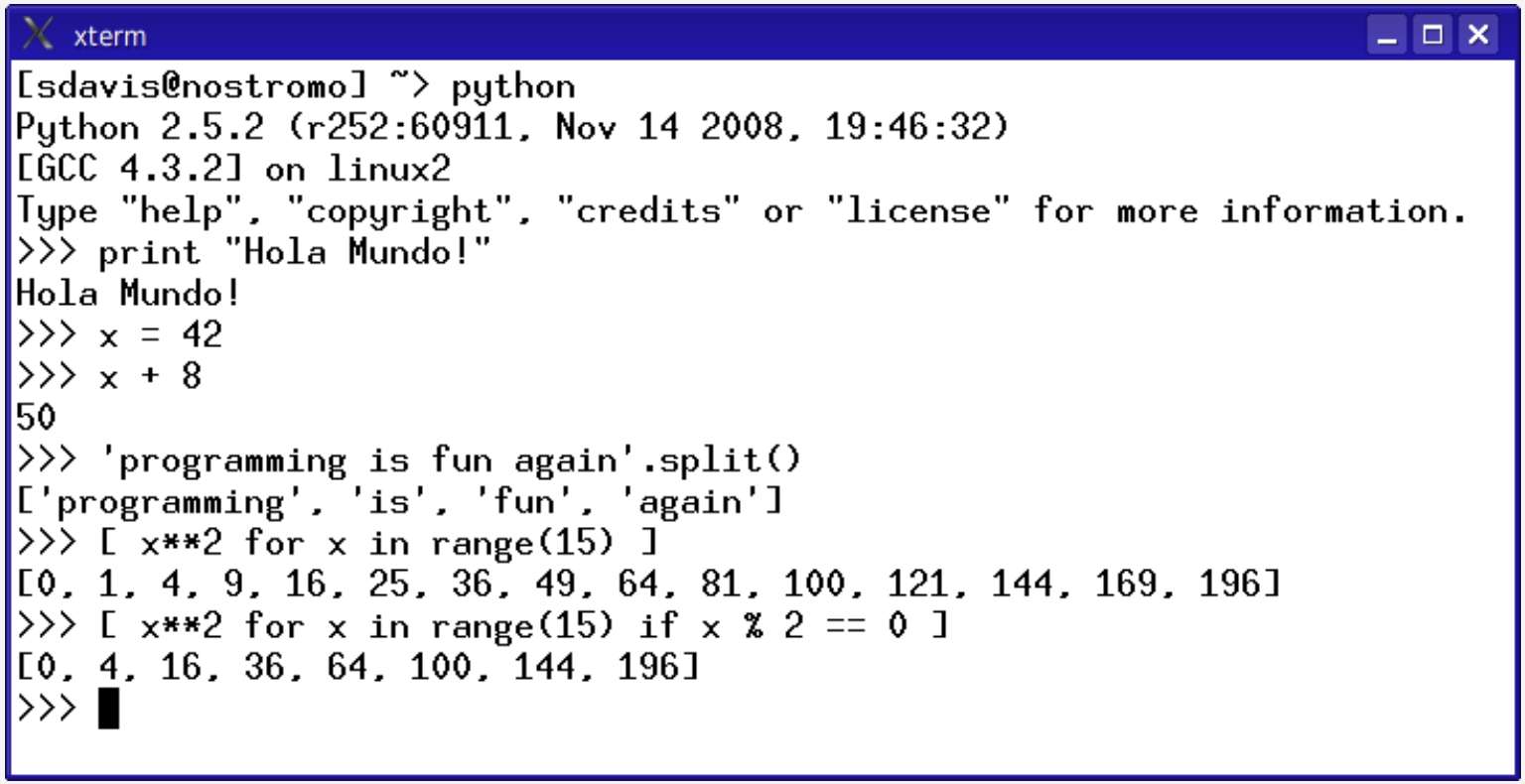
¿Cómo ejecutar programas en Python?

Existen dos formas:

- Dentro del intérprete interactivo. Esto es útil para probar pequeñas instrucciones, depurar programas, o buscar ayuda de funciones y métodos.
- Como un programa o script independiente, en caso de un programa en su forma final o que ya tenga definidas funciones y clases propias.

Dentro del intérprete interactivo

Simplemente llamando al comando `python` desde la línea de comandos:



¿Cómo funciona el modo interactivo?

- El *prompt* >>> indica que Python está listo para recibir instrucciones
- Si uno tipea una expresión, esta se evalúa y se muestra directamente su resultado
- La ayuda para una función se obtiene con `help` y el nombre de la función (entre paréntesis)

```
>>> help(float)
```

- La ayuda para una palabra clave se obtiene con `help` y la palabra (entre comillas simples y paréntesis)

```
>>> help('while')
```

- Llamar a `help()` (sin argumentos) accede a la ayuda interactiva

Como un programa o *script* independiente

```
#!/usr/bin/env python

# Este es un programa o script en Python
# En una linea, todo lo que sigue a
# continuacion de un caracter # es comentario

print "Hola, Mundo!"

x = 42
print x + 8              # imprime 50

print 'programming is fun again'.split()

print [x**2 for x in range(15)]
print [x**2 for x in range(15) if x % 2 == 0]
```

La línea #!/usr/bin/env python es típica de un script en UNIX

Una vez más... ¡Cuidado con la indentación!

Es muy común para el que comienza en Python olvidar que el espacio en blanco al inicio es importante. Por ejemplo:

```
# Este es el tipico error que uno comete
# cuando aprende Python!
print "Hola Mundo!"
 print "Este es mi primer programa!"
```

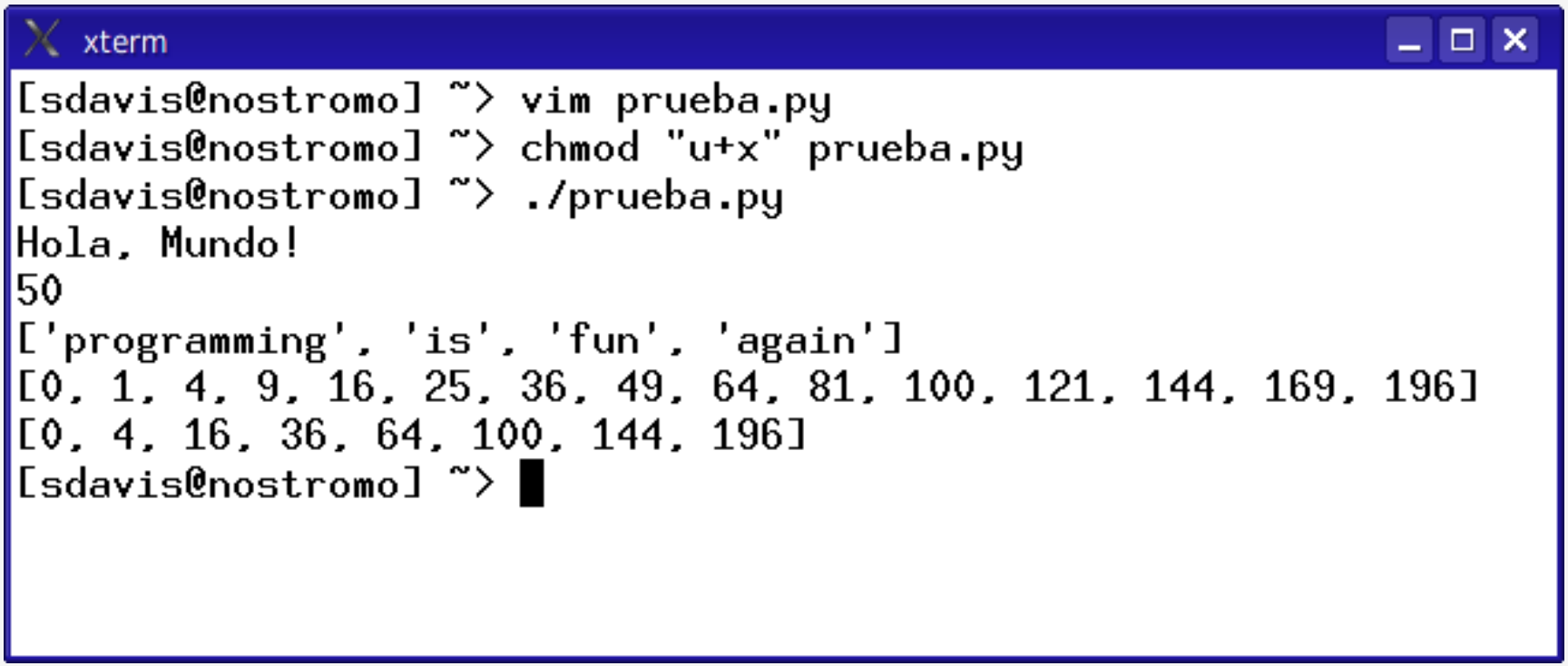
Ejemplo: Un programa de prueba

Al correr el programa anterior, Python dirá:

```
  File "primero.py", line 4
    print "Este es mi primer programa!"
    ^
IndentationError: unexpected indent
```

Python sólo espera un cambio de indentación después de los dos puntos a continuación de un bloque `if`, `for`, `while`, `def`, `class`, etc.

Ejecutando un programa



Ahora... a la práctica!

1. Hacer una copia del directorio `/home/sdavis/tallerpython` a su propio `/home`
2. En el directorio `tallerpython/charla1` se encuentran programas sencillos de ejemplo:

prueba.py	El programa de prueba visto anteriormente
fibo.py	Calcula los términos de la secuencia de Fibonacci menores que 1000
plot.py	Dibuja $\sin(x)$ y $\cos(x)$ en una ventana gráfica
calcpi.py	Calcula $\pi = 3{,}141592653589...$ usando una expansión en series

La idea es sentirse cómodo editando y ejecutando estos ejemplos, tratar de entender la idea general y la sintaxis, y comenzar a experimentar desde ya creando programas propios, mezclando ideas de los ejemplos.