

PYTHON



**Universidad Nacional de
Ingenieria**

Facultad de Ciencias

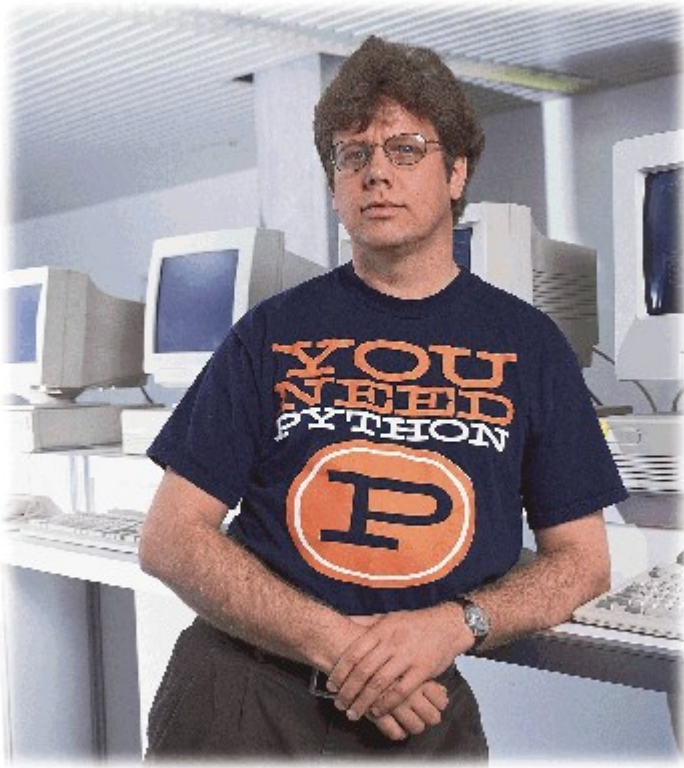
Abraham Zamudio Ch.

abraham.zamudio@gmail.com

PYTHON

Clase 1

Guido Van Rossum



En el ambiente de los desarrolladores del lenguaje Python también se le conoce por el título BDFL (Benevolent Dictator for Life), teniendo asignada la tarea de fijar las directrices sobre la evolución de Python, así como la de tomar decisiones finales sobre el lenguaje que todos los desarrolladores acatan. Guido tiene fama de ser bastante conservador, realizando pocos cambios al lenguaje entre versiones sucesivas, intentando mantener siempre la compatibilidad con versiones anteriores.

Algunos usuarios de PYTHON



Python es lo suficientemente veloz para nuestro sitio y nos permite producir características mantenibles en tiempo récord con un mínimo de desarrolladores"

Cuong Do, Software Architect, YouTube.com.

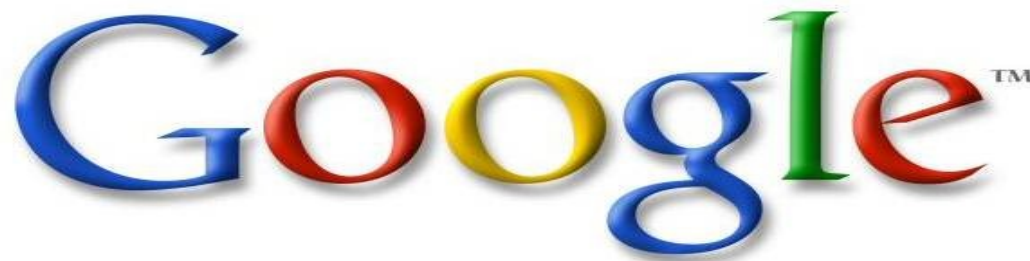
Algunos usuarios de PYTHON



"Python juega un rol clave en nuestra línea de producción. Sin él, un proyecto del tamaño de Star Wars: Episode II hubiera sido muy difícil de realizar. Desde la renderización de multitudes, al procesamiento por lotes, a la composición, Python une todas estas cosas juntas,"

Tommy Burnette, Senior Technical Director, ILM.

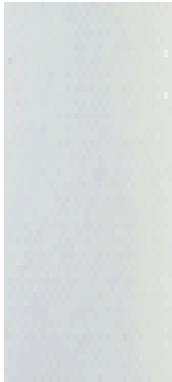
Algunos usuarios de PYTHON



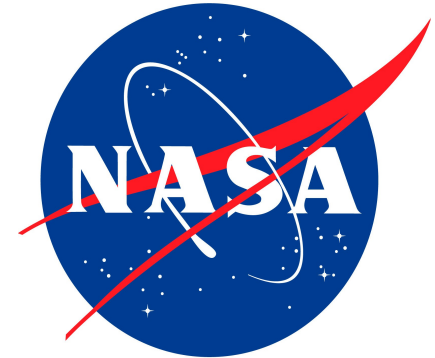
"Python ha sido una parte importante de Google desde el comienzo, y lo sigue siendo mientras el sistema crece y evoluciona. Hoy docenas de ingenieros de Google que usan Python, y estamos buscando más personas con habilidades en este lenguaje."

**Peter Norvig, Director of Search Quality
Google, Inc.**

Algunos usuarios de PYTHON



CIRANOVA



Radio Observatorio de
JICAMARCA
Radio Observatory



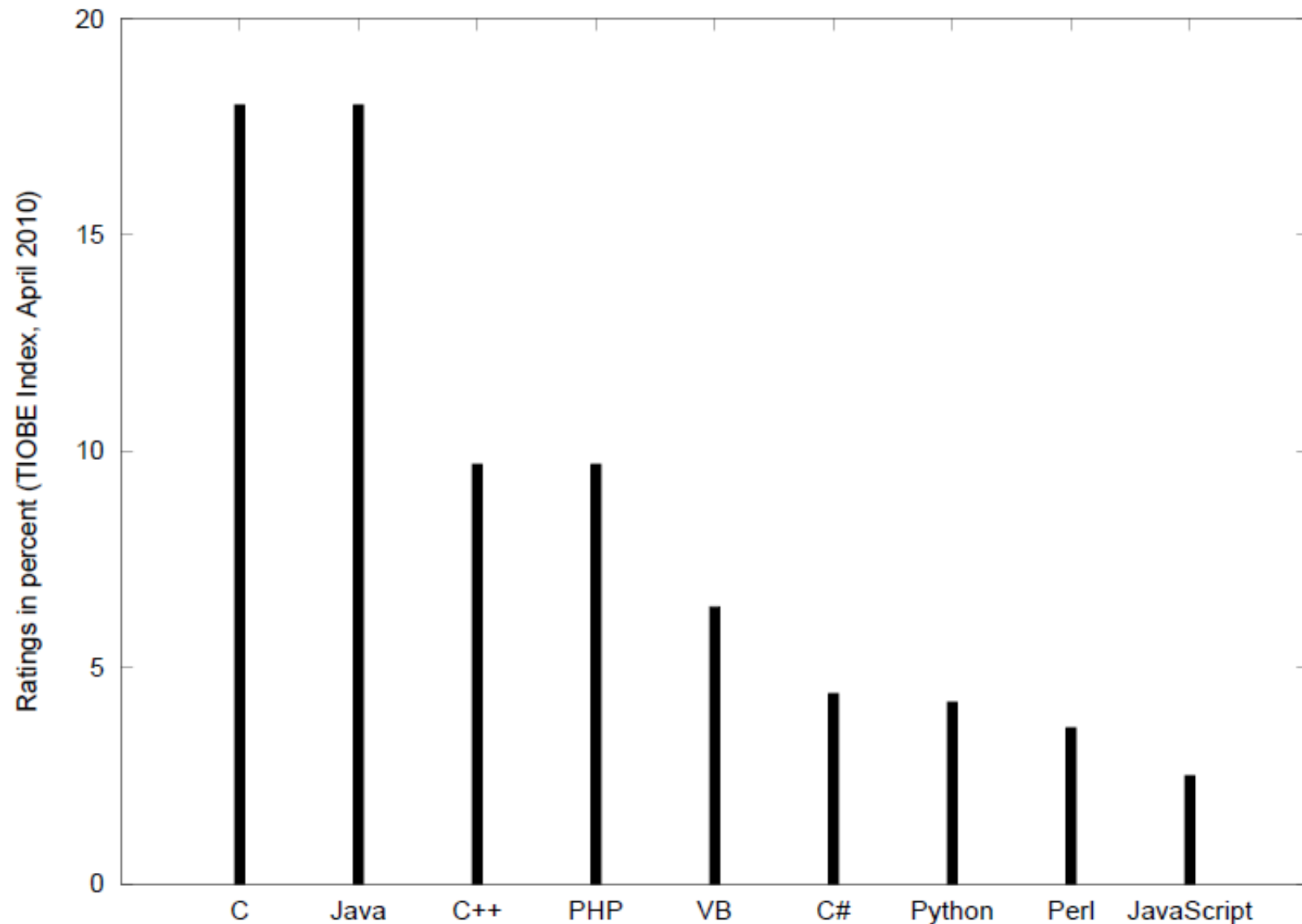
YAHOO! GROUPS

Por que usar Python

Python is a dynamic object-oriented programming language that can be used for many kinds of software development. It offers strong support for integration with other languages and tools, comes with extensive standard libraries, and can be learned in a few days. Many Python programmers report substantial productivity gains and feel the language encourages the development of higher quality, more maintainable code.

— python.org

Popularidad de los Lenguajes de Programacion



Python es un muy BUEN entorno de Programacion

Very clean syntax,
high-level statements,
“executable pseudo code”

```
def myfunc(x, y, t):  
    return sinh(x)*cosh(y)*exp(-0.15*t)  
  
t = 0  
while t < T:  
    A, b = matrixfactory(grid, u, myfunc)  
    P, status = ML.preconditioner(A)  
    x = linear_solver.(A, b, M)  
    u.set_new_values(x)  
    vtk.visualize(u, t)  
    netcdf.store(u, t); pickle.dump(u)  
    GUI.update(t)  
    t += dt
```

Python es un muy BUEN entorno de Programacion

Variables can hold objects of any type

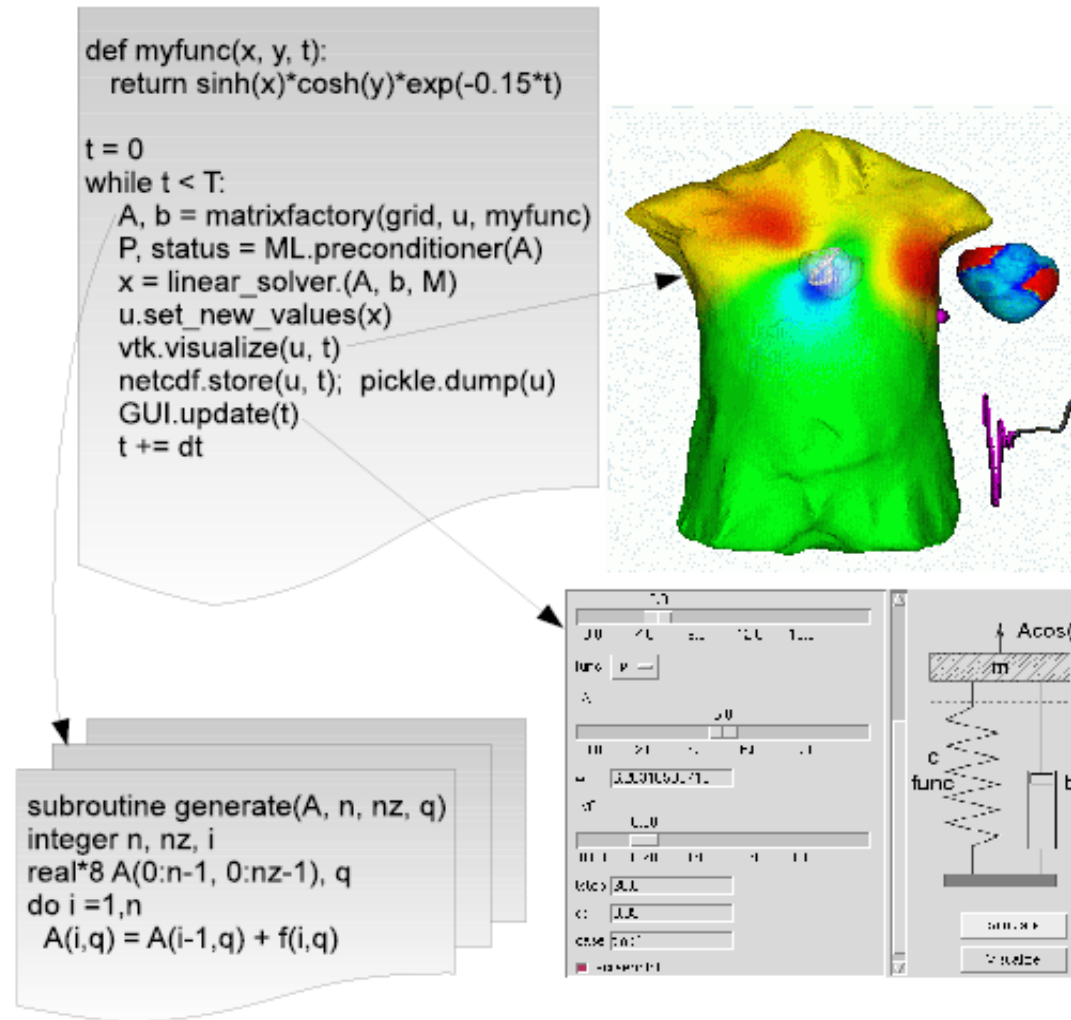
```
def myfunc(x, y, t):  
    return sinh(x)*cosh(y)*exp(-0.15*t)  
  
t = 0  
while t < T:  
    A, b = matrixfactory(grid, u, myfunc)  
    P, status = ML.preconditioner(A)  
    x = linear_solver.(A, b, M)  
    u.set_new_values(x)  
    vtk.visualize(u, t)  
    netcdf.store(u, t); pickle.dump(u)  
    GUI.update(t)  
    t += dt
```

Python es un muy BUEN entorno de Programacion

```
def myfunc(x, y, t):  
    return sinh(x)*cosh(y)*exp(-0.15*t)  
  
t = 0  
while t < T:  
    A, b = matrixfactory(grid, u, myfunc)  
    P, status = ML.preconditioner(A)  
    x = linear_solver.(A, b, M)  
    u.set_new_values(x)  
    vtk.visualize(u, t)  
    netcdf.store(u, t); pickle.dump(u)  
    GUI.update(t)  
    t += dt
```

Program or
interactive session

Python es un muy BUEN entorno de Programacion

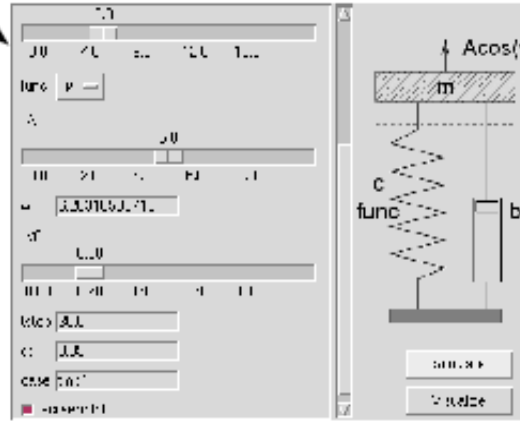
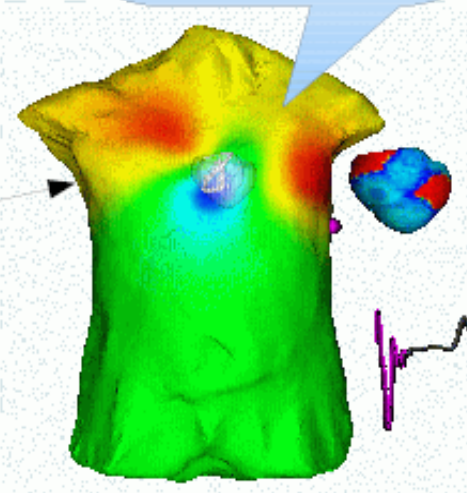


Python es un muy BUEN entorno de Programacion

```
def myfunc(x, y, t):  
    return sinh(x)*cosh(y)*exp(-0.15*t)  
  
t = 0  
while t < T:  
    A, b = matrixfactory(grid, u, myfunc)  
    P, status = ML.preconditioner(A)  
    x = linear_solver(A, b, M)  
    u.set_new_values(x)  
    vtk.visualize(u, t)  
    netcdf.store(u, t); pickle.dump(u)  
    GUI.update(t)  
    t += dt
```

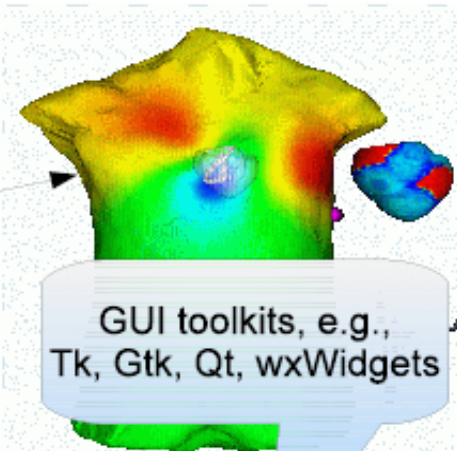
Visualization, e.g.,
Vtk, OpenDX, ...

```
subroutine generate(A, n, nz, q)  
    integer n, nz, i  
    real*8 A(0:n-1, 0:nz-1), q  
    do i = 1, n  
        A(i, q) = A(i-1, q) + f(i, q)
```

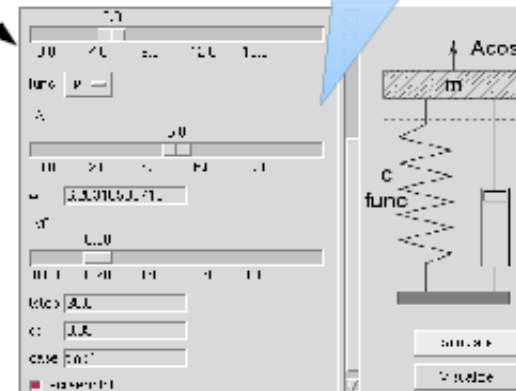


Python es un muy BUEN entorno de Programacion

```
def myfunc(x, y, t):  
    return sinh(x)*cosh(y)*exp(-0.15*t)  
  
t = 0  
while t < T:  
    A, b = matrixfactory(grid, u, myfunc)  
    P, status = ML.preconditioner(A)  
    x = linear_solver(A, b, M)  
    u.set_new_values(x)  
    vtk.visualize(u, t)  
    netcdf.store(u, t); pickle.dump(u)  
    GUI.update(t)  
    t += dt
```



```
subroutine generate(A, n, nz, q)  
    integer n, nz, i  
    real*8 A(0:n-1, 0:nz-1), q  
    do i = 1, n  
        A(i, q) = A(i-1, q) + f(i, q)
```

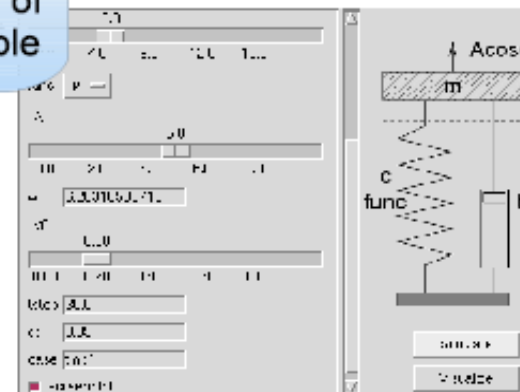
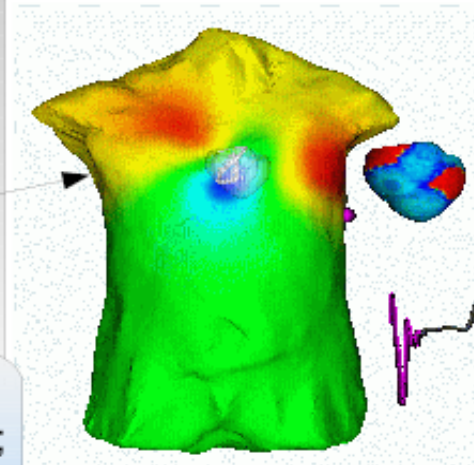


Python es un muy BUEN entorno de Programacion

```
def myfunc(x, y, t):  
    return sinh(x)*cosh(y)*exp(-0.15*t)  
  
t = 0  
while t < T:  
    A, b = matrixfactory(grid, u, myfunc)  
    P, status = ML.preconditioner(A)  
    x = linear_solver(A, b, M)  
    u.set_new_values(x)  
    vtk.visualize(u, t)  
    netcdf.store(u, t); pickle.dump(u)  
    GUI.update(t)  
    t += dt
```

Easy interfacing to
Fortran, C, C++ codes;
great simplification of
interfaces is possible

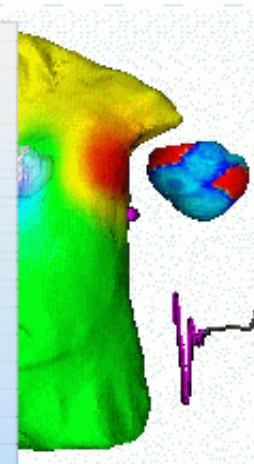
```
subroutine generate(A, n, nz, q)  
    integer n, nz, i  
    real*8 A(0:n-1, 0:nz-1), q  
    do i = 1, n  
        A(i, q) = A(i-1, q) + f(i, q)
```



Python es un muy BUEN entorno de Programacion

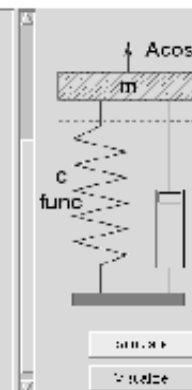
```
def myfunc(x, y, t):  
    return sinh(x)*cosh(y)*exp(-0.15*t)
```

- Matlab-ish arrays and array computing
- Flexible data structures (hash, list, class)
- Great software engineering support (packages, unit tests, documentation)
- Python scales from small sessions/scripts to large software systems
- Rich standard library
- Wide collection of third-party modules
- Cross-platform operating system interface
- Support of all major programming styles
- Overloaded operators
- I/O tools (pickle, shelve, netCDF, HDF5)
- Regular expressions for text processing
- Run-time code generation
- Free, open source



```
su  
int  
re  
do i = 1,n  
    A(i,q) = A(i-1,q) + 1/(i,q)
```

```
0.0  
0.1 0.2 0.3 0.4 0.5  
t0: 0.0  
c: 0.0  
case: 0.0  
-12 0.0 0.0 0.0
```



¿Que es y algunas características ?

Python es un lenguaje de programación interpretado, interactivo y orientado a objetos. Incorpora módulos, excepciones, tipado dinámico, tipos de datos dinámicos de muy alto nivel, y clases. Python combina un remarcable poder con una sintáxis muy clara.



Sintaxis elegante, minimalista y densa

En Python todo aquello innecesario no hay que escribirlo (;, {, }, \n). Además como veremos pocas líneas de código revierten en mucha funcionalidad.



Ejemplo : Copia de archivo

- `#include <stdio.h>`
- `int main(int argc, char **argv) {`
- `FILE *in, *out;`
- `int c;`
- `in = fopen("input.txt", "r");`
- `out = fopen("output.txt", "w");`
- `while ((c = fgetc(in)) != EOF) {`
- `fputc(c, out);`
- `}`
- `fclose(out);`
- `fclose(in);`
- `}`
- `in = open("input.txt")`
- `out = open("output.txt", "w")`
- `out.writelines(in)`
- `out.close()`



Ejemplo : Suma de numeros aleatorios negativos

```
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
```

```
# 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;
}
```

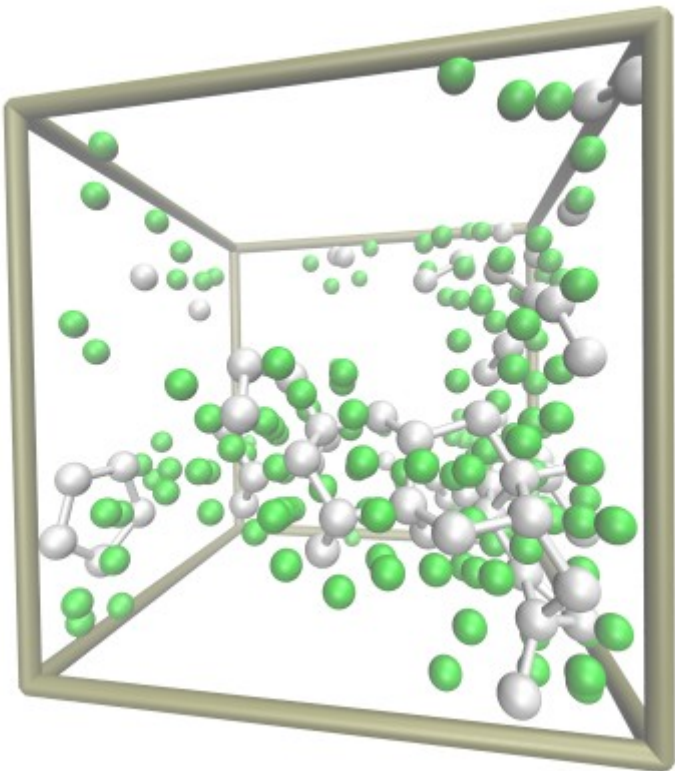
Ejemplo : Suma de numeros aleatorios negativos

```
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 (version estructurada)

Ejemplo : Suma de numeros aleatorios negativos

```
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 (version funcional)

Moderno

Soporta objetos y estructuras de datos de alto nivel tales como cadenas de caracteres (strings), listas, diccionarios.



Ejemplo: Secuencia de Numeros

```
public class ConsoleTest {  
    public static void main(String[]  
        args) {  
        for (int i = 0; i < 1000000; i++) {  
            System.out.println(i);  
        }  
    }  
}
```

```
for x in xrange(1000000):  
    print x
```



Multi-paradigma

En vez de forzar a los programadores a adoptar un paradigma de programación único

```
def add5(x):
    return x+5

def dotwrite(ast):
    nodename = getNodeName()
    label=symbol.sym_name.get(int(ast[0]),ast[0])
    print '    %s [label="%s' % (nodename, label),
    if isinstance(ast[1], str):
        if ast[1].strip():
            print '= %s";' % ast[1]
        else:
            print '"]'
    else:
        print '["];'
        children = []
        for n, child in enumerate(ast[1:]):
            children.append(dotwrite(child))
        print ', ' %s -> {' % nodename
        for n, child in enumerate(children):
            print '%s' % name,
```

Python permite el uso de programación orientada a objetos, programación estructurada o procedural e incluso programación funcional.

Organizado y extendible

Dispone de múltiples formas de organizar código tales como funciones, clases, módulos, y paquetes. Si hay áreas que son lentas se pueden reemplazar por plugins (extensiones o wrappers) en C o C++, siguiendo la API para extender o empotrar Python en una aplicación.



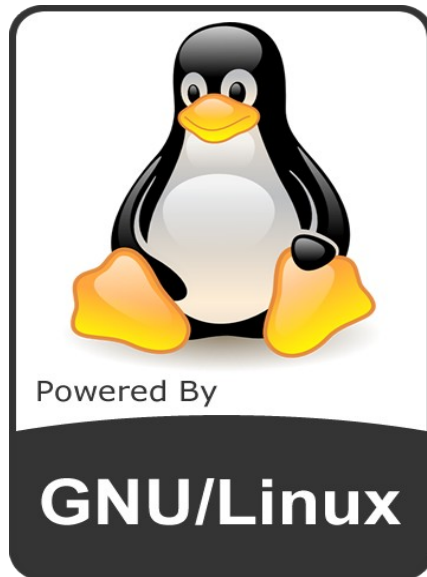
Interpretado

No es necesario declarar constantes y variables antes de utilizarlas y no requiere paso de compilación/linkaje. La primera vez que se ejecuta un script de Python se compila y genera bytecode que es luego interpretado. Python es dinámico, encuentra errores de uso de tipos de datos en tiempo de ejecución y usa un recolector de basura para la gestión de memoria.

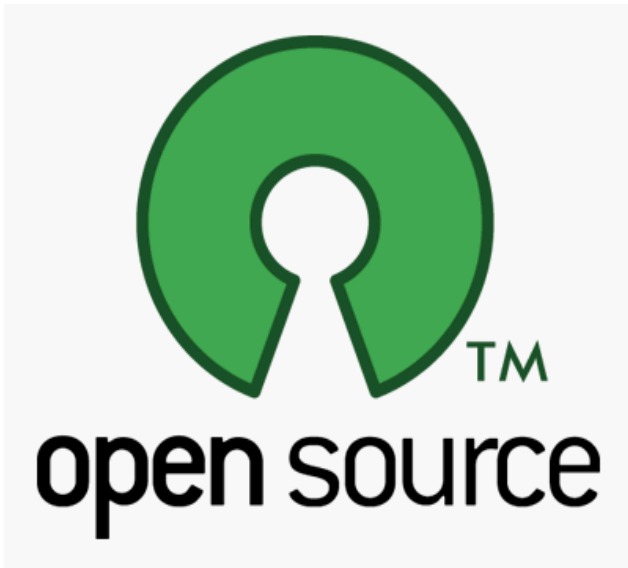


Multiplataforma

Python genera código interoperable, que se puede ejecutar en múltiples plataformas (más aún que Java). Además, es posible embeber Python dentro de una JVM



Open source

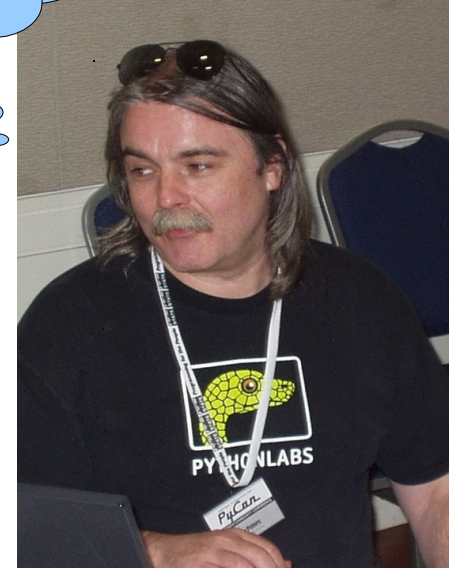


Razón por la cual la librería de módulos de Python (<http://docs.python.org/lib/lib.html>) contiene un sinfín de módulos de utilidad y sigue creciendo continuamente.

Zen de Python

Enumeración de los principios de diseño y la filosofía de Python útiles para comprender y utilizar el lenguaje.

Tim
Peters



- Bello es mejor que feo.
- Explícito es mejor que implícito.
- Simple es mejor que complejo.
- Complejo es mejor que complicado.
- Plano es mejor que anidado.
- Ralo es mejor que denso.
- La legibilidad cuenta.
- Los casos especiales no son tan especiales como para quebrantar las reglas.
- Aunque lo práctico gana a la pureza.
- Los errores nunca deberían dejarse pasar silenciosamente.
- A menos que hayan sido silenciados explícitamente.
- Frente a la ambigüedad, rechaza la tentación de adivinar.
- Debería haber una -y preferiblemente sólo una- manera obvia de hacerlo.
- Aunque esa manera puede no ser obvia al principio a menos que usted sea holandés.
- Ahora es mejor que nunca.
- Aunque nunca es a menudo mejor que ya mismo.
- Si la implementación es difícil de explicar, es una mala idea.
- Si la implementación es fácil de explicar, puede que sea una buena idea.
- Los espacios de nombres (namespaces) son una gran idea ¡Hagamos más de esas cosas!

Abraham Zamudio

31

(la version completa en ingles aparece con
`import this`)

La biblioteca Standard de python

Python tiene una gran biblioteca estándar, usada para una diversidad de tareas. Esto viene de la filosofía "pilas incluidas" ("batteries included") en referencia a los módulos de Python. Los módulos de la biblioteca estándar pueden mejorarse por módulos personalizados escritos tanto en C como en Python. Debido a la gran variedad de herramientas incluidas en la biblioteca estándar combinada con la habilidad de usar lenguajes de bajo nivel como C y C++, los cuales son capaces de interactuar con otras bibliotecas, Python es un lenguaje que combina su clara sintaxis con el inmenso poder de lenguajes menos elegantes.

Las pilas puestas

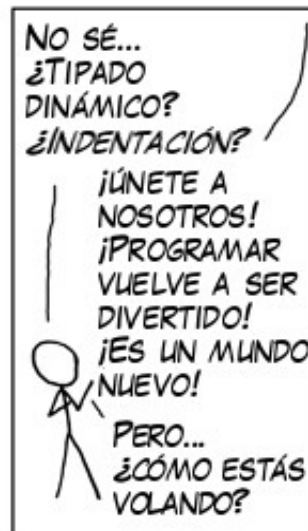
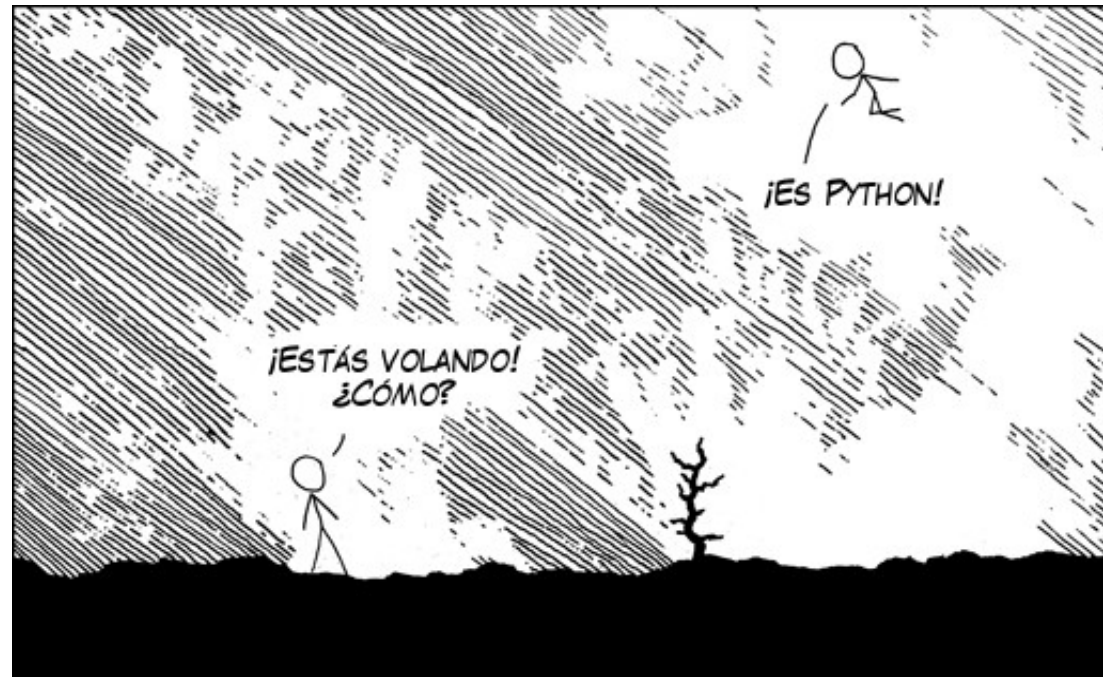
Servicios del sistema, fecha y hora, subprocesos, sockets, internacionalización y localización, base de datos, threads, formatos zip, bzip2, gzip, expresiones regulares, XML (DOM y SAX), Unicode, SGML, HTML, XHTML, email, manejo asíncrono de sockets, clientes HTTP, FTP, SMTP, NNTP, POP3, IMAP4, servidores HTTP, SMTP, herramientas MIME, interfaz con el garbage collector, serializador y deserializador de objetos, debugger, profiler, random, curses, logging, compilador, decompilador, CSV, análisis lexicográfico, interfaz gráfica incorporada, matemática real y compleja, criptografía, introspección, unit testing, doc testing, etc., etc...



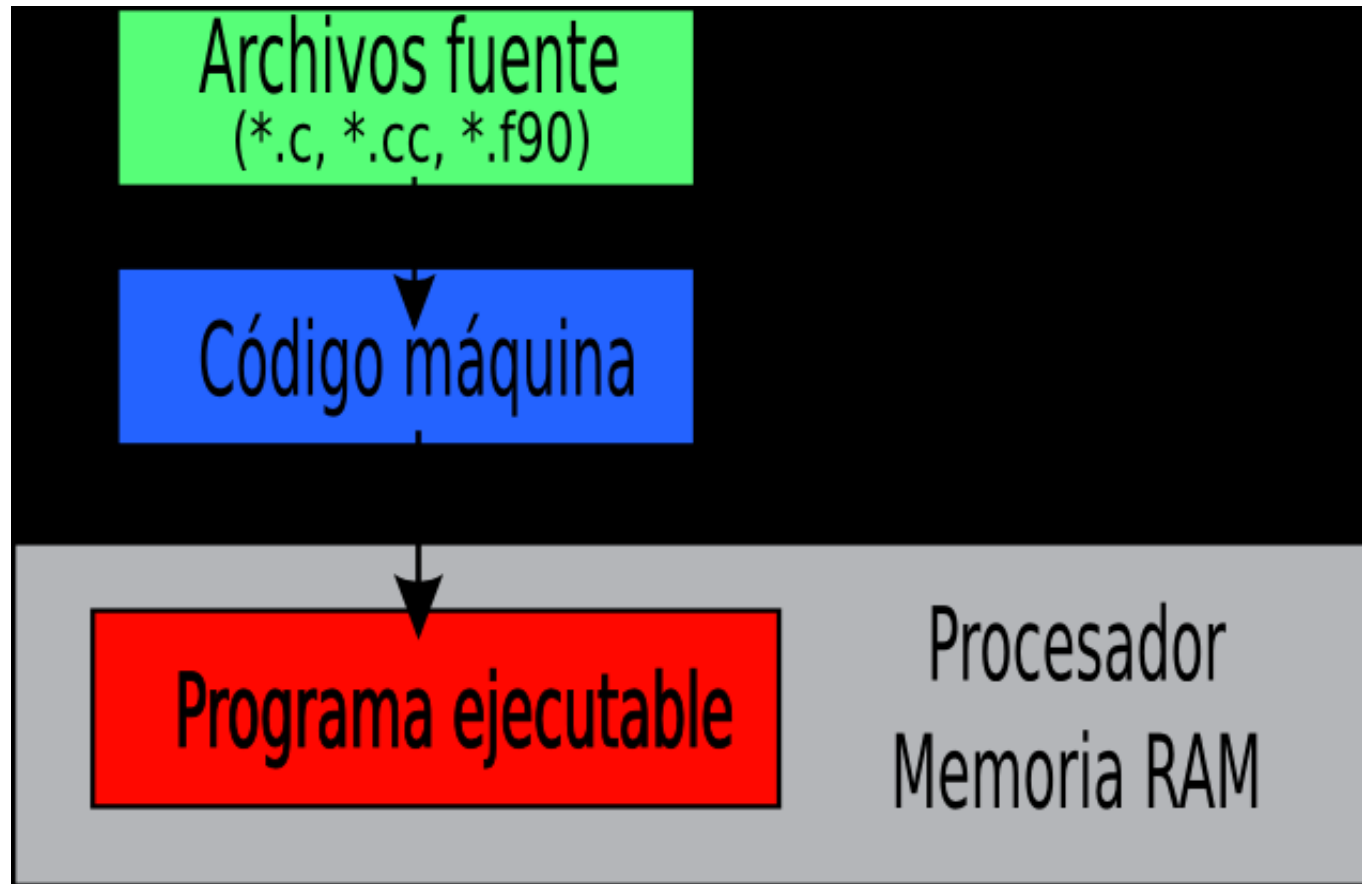
Algunas habilidades

- Bases de datos
 - MySQL, PostgreSQL, MS SQL, Informix, DB/2, Sybase
- Interfaces gráficas
 - Qt, GTK, win32, wxWidgets, Cairo
- Frameworks Web
 - Django, Turbogears, Zope, Plone, webpy
- Varios mas
 - PIL: Python Imaging Library
 - Matplotlib es una biblioteca para la generación de gráficos a partir de datos contenidos en listas o arrays en el lenguaje de programación Python y su extensión matemática NumPy. Proporciona una API, pylab, diseñada para recordar a la de MATLAB.
 - Scipy: Scientific Python

Con Python, programar vuelve a ser divertido! (XKCD comic numero 353, "Python")

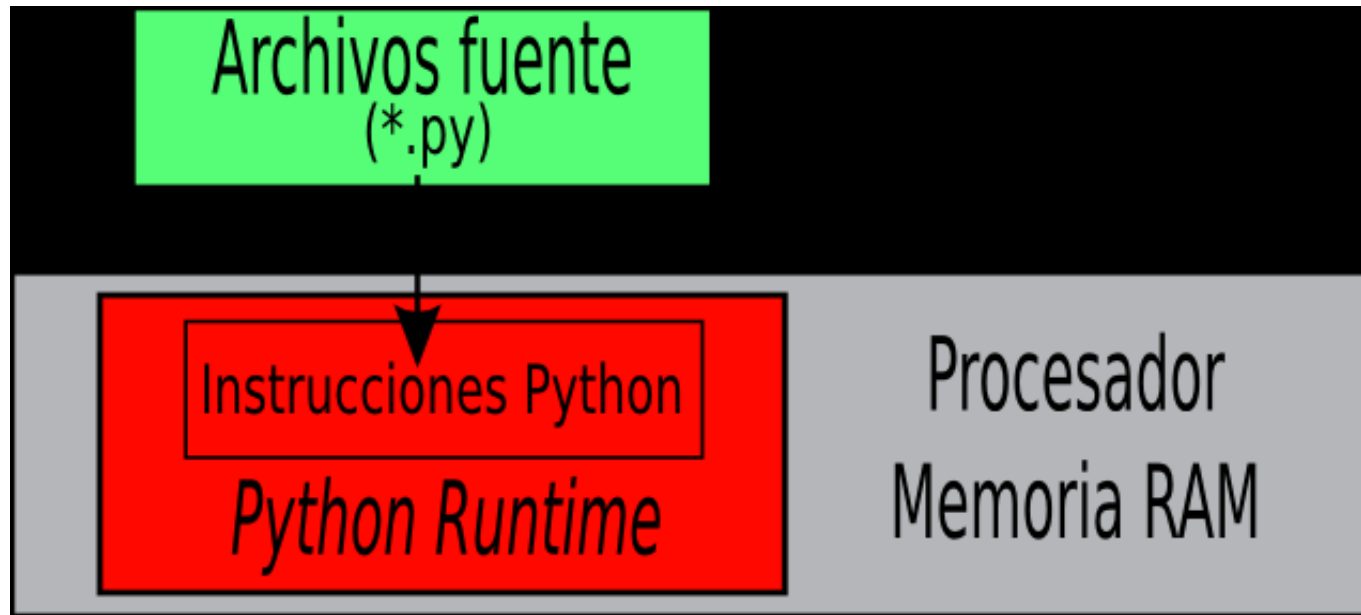


¿Como funciona un lenguaje compilado?



- 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.

¿Como 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 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 el) 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 mas comoda es usar un contenedor, llenandolo 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...

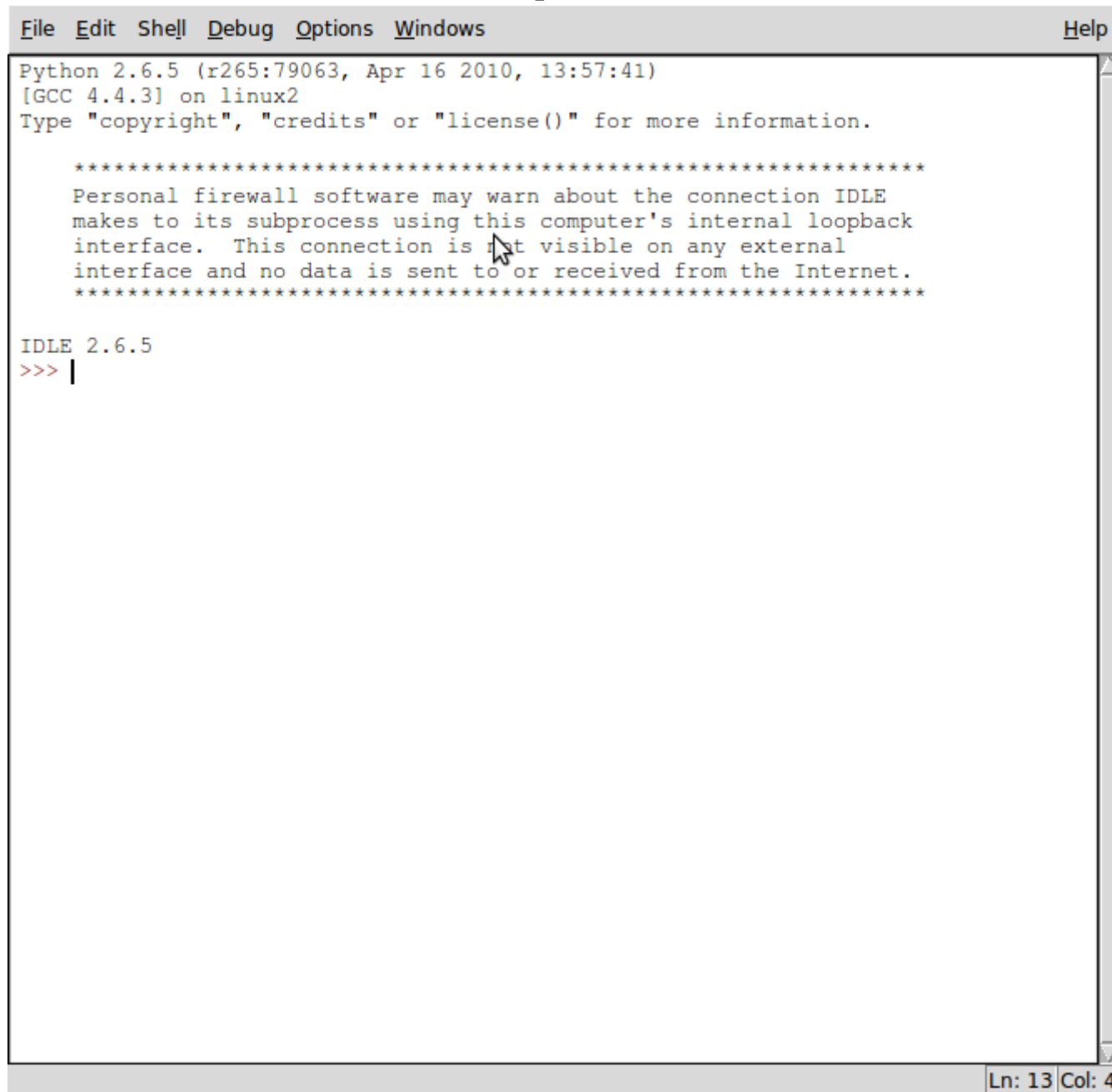
- No más fugas de memoria (*memory leaks*)
- No más violaciones de segmento (*segmentation faults*)

¿Como ejecutar programas en Python?

Existen dos formas:

- Dentro del interprete interactivo. Esto es util para probar pequeñas instrucciones, depurar programas, o buscar ayuda de funciones y metodos.
- 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 interprete interactivo



```
File Edit Shell Debug Options Windows Help
Python 2.6.5 (r265:79063, Apr 16 2010, 13:57:41)
[GCC 4.4.3] on linux2
Type "copyright", "credits" or "license()" for more information.

*****
Personal firewall software may warn about the connection IDLE
makes to its subprocess using this computer's internal loopback
interface. This connection is not visible on any external
interface and no data is sent to or received from the Internet.
*****

IDLE 2.6.5
>>> |
```

Ln: 13 Col: 4

Dentro del interprete interactivo

- El prompt `>>>` indica que Python esta listo para recibir instrucciones
- Si uno tipea una expresion, esta se evalua y se muestra directamente su resultado
- La ayuda para una funcion se obtiene con `help` y el nombre de la funcion (entre parentesis)

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

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

```
>>> 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 linea `#!/usr/bin/env python` es tipica de un script en UNIX

Una vez mas... ¡Cuidado con la indentacion!

Es muy comun 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 dira:

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

Python Basico (Calculadora)

Enteros

```
>>> 2+2
```

```
4
```

```
>>> (50 - 5*6) / 4
```

```
5
```

```
>>> 7 / 3
```

```
2
```

```
>>> 7 % 3 (Operador Modulo)
```

```
1
```

```
>>> 23098742098472039 * 120894739
```

```
2792516397223089453702821
```

Floats

```
>>> 3 * 3.75 / 1.5
```

```
7.5
```

```
>>> 7 / 2.3
```

```
3.0434782608695654
```

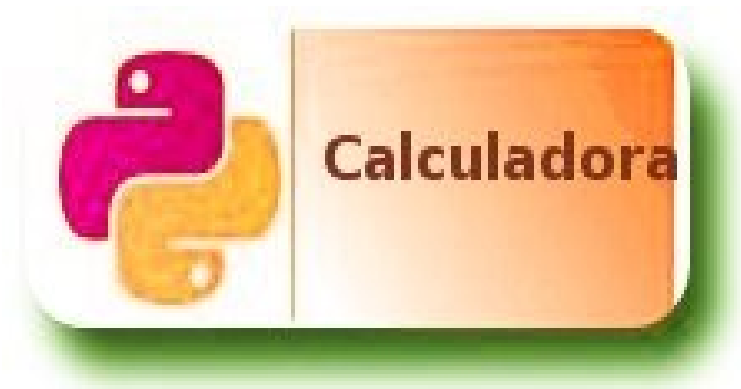
Algo mas

Complejos

```
>>> 2 + 3j
(2+3j)
>>> (2+3j) * (2+5j)
(-11+16j)
>>> (3-4j) ** 2
(-7-24j)
```

Recortando los decimales

```
>>> int(12.3)
12
>>> round(2.7526)
3.0
>>> round(2.7526, 2)
2.75
```



Tipos de Datos: Tuplas y Listas

Para declarar una lista, basta usar los corchetes [], mientras que para declarar una tupla es recomendable usar los paréntesis (). En ambas los elementos se separan por comas.

- Tanto las listas como las tuplas pueden contener elementos de diferentes tipos. No obstante las listas suelen usarse para elementos del mismo tipo en cantidad variable mientras que las tuplas se reservan para elementos distintos en cantidad fija.
- Para acceder a los elementos de una lista o tupla, se utiliza un índice entero. Se pueden utilizar índices negativos para acceder elementos a partir del final.
- Las listas se caracterizan por ser mutables, es decir, se puede cambiar su contenido, mientras que no es posible modificar el contenido de una tupla ya creada, puesto que es inmutable.

Ejemplos : Tuplas y listas

```
>>> lista = ['aaa', 1, 90]
```

```
>>> lista[-1]
```

```
90
```

```
>>> lista[0] = 'xyz'
```

```
>>> lista[0:2]
```

```
['xyz', 1]
```

```
>>> tupla = (1, 2, 3)
```

```
>>> tupla[0] = 2
```

```
( genera un error )
```

```
>>> tupla[0]
```

```
1
```

```
>>> otratupla = (tupla, ('a', 'b')) # es posible anidar tuplas
```

Tipos de Datos: Diccionarios

Los diccionarios se declaran entre llaves ({}), y contienen elementos separados por comas, donde cada elemento está formado por un par clave : valor (el símbolo : separa la clave de su valor correspondiente).

Ejemplo de como llamamos a valores en un diccionario mediante su clave. La clave la pondremos entre un corchete abierto "[" y un corchete cerrado "]". Esto nos devolverá el valor.

```
>>> mi_diccionario = {"coche": "rojo", "mesa": "marron"}
>>> mi_diccionario['coche']
'rojo'
>>> mi_diccionario['mesa']
'marron'
```

Los diccionarios son mutables:

```
>>> mi_diccionario = {"numero_globos": 4}
>>> mi_diccionario["numero_globos"]
4
>>> mi_diccionario["numero_globos"] = 3
>>># Podemos reasignar el valor de una clave
>>> mi_diccionario["numero_globos"]
3
```

El valor de un diccionario puede ser de cualquier tipo. Un string ("casa"), un int (2), una lista ([1, 2, 3]), o incluso un diccionario.

```
>>> mi_diccionario = {"coches": 4, "tipo": "de los que tienen 4 ruedas", "marcas":
['Honda', 'Renault', 'Seat'], "descripciones": {"uno": ('Seat', 'Ibiza', 'rojo'), "dos": ('Honda',
'civic', 'azul'), "tres": ('Seat', 'Ibiza', 'negro')} }
>>> mi_diccionario['coches']
4
>>> mi_diccionario['tipo']
'de los que tienen 4 ruedas'
>>> mi_diccionario['marcas']
['Honda', 'Renault', 'Seat']
>>> mi_diccionario['marcas'][0]
'Honda'
>>> mi_diccionario['descripciones']
{"uno": ('Seat', 'Ibiza', 'rojo'), "dos": ('Honda', 'civic', 'azul'), "tres": ('Seat', 'Ibiza', 'negro')}
>>> mi_diccionario['descripciones']['uno']
('Seat', 'Ibiza', 'rojo')
```

En cambio, la clave en los diccionarios debe ser inmutable. Esto quiere decir, por ejemplo, que no podremos usar ni listas ni diccionarios como claves:

```
>>> mi_diccionario = {"coche": 'rojo'} # Con un string. Valido.  
>>> mi_diccionario = {4: "manuales"} # Con un int. Valido.  
>>> mi_diccionario = {(2, 'lampara'): "florero"} # Con una  
tupla. Valido.  
>>> mi_diccionario = {[2, 'lampara']: "florero"} # Con una lista.  
Error.  
(... Error ...)  
>>> mi_diccionario = {'fuente': 'luz': "dimm"} # Con un  
diccionario. Error.  
(... Error ...)
```

Asignacion de Variables

Referencia a una variable sin asignar

```
>>> x
```

```
Traceback (innermost last):
```

```
  File "<interactive input>", line 1, in ?
```

```
NameError: There is no variable named 'x'
```

```
>>> x = 1
```

```
>>> x
```

```
1
```

Asignación de múltiples valores simultáneamente

```
>>> v = ('a', 'b', 'e')
```

```
>>> (x, y, z) = v
```

```
>>> x
```

```
'a'
```

```
>>> y
```

```
'b'
```

```
>>> z
```

```
'e'
```

Asignación de valores consecutivos

```
>>> range(7)
[0, 1, 2, 3, 4, 5, 6]
>>> (LUNES, MARTES, MIERCOLES, JUEVES,
VIENRES, SABADO, DOMINGO) = range(7)
>>> LUNES
0
>>> MARTES
1
>>> DOMINGO
6
```


Entrada y Salida de Datos

Funcion print :

```
>>>a, saludo = 5, 'Hola'
```

```
>>>print 'a contiene el valor' , a, 'y salud contiene el valor', saludo
```

```
>>> print int(2.9)
```

```
2
```

```
>>> print float(2)
```

```
2.0
```

```
>>> print float(2/3)
```

```
0.0
```

Funcion raw_input()

```
>>>print '¿Cómo te llamas?'  
>>>nombre = raw_input()  
>>>print 'Me alegro de conocerte,' , nombre
```

Funcion raw_input()

```
nombre = raw_input('¿Cómo te llamas?')
```

```
print 'Me alegro de conocerte,' , nombre
```

Por defecto, la función `raw_input()` convierte la entrada en una cadena. Si quieres que Python interprete la entrada como un número entero, debes utilizar la función `int()` de la siguiente manera:

```
>>>print 'Dime una cantidad en soles: ',  
>>>cantidad = int(raw_input())  
>>>print cantidad, 'Dolares son' , cantidad/2,86, '$'
```

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 indentacion en Python

En Python, la cantidad de espacio en blanco al inicio de la linea (llamado el nivel de indentacion) es lo unico 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.

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();
    }
}
```

¡Python tiene un unico estilo! (o un estilo unico...)

```
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 ambigüedades
- Un programador ordenado de todas maneras agrega espacios para hacer el código más legible

Control de flujo

Sentencias condicionales

Si un programa no fuera más que una lista de órdenes a ejecutar de forma secuencial, una por una, no tendría mucha utilidad. Los condicionales nos permiten comprobar condiciones y hacer que nuestro programa se comporte de una forma u otra, que ejecute un fragmento de código u otro, dependiendo de esta condición.

Sentencia : if

La forma más simple de un estamento condicional es un if (del inglés si) seguido de la condición a evaluar, dos puntos (:) y en la siguiente línea e indentado, el código a ejecutar en caso de que se cumpla dicha condición.

```
fav = "mundogeek.net"  
if fav == "mundogeek.net":  
    print "Tienes buen gusto!"  
    print "Gracias"
```

Tu_edad.py

```
#!/usr/bin/env python
edad = int(raw_input('Cuantos anhos tienes '))
if edad >= 90:
    print "Tienes 90 anhos o mas..."
else:
    if edad <= 0:
        print "Has metido una edad no valida"
    else:
        if edad < 18:
            print 'Eres menor de edad'
        else:
            print 'Eres mayor de edad'
print 'Hasta la proxima'
raw_input()
```

if ... else

Vamos a ver ahora un condicional algo más complicado. ¿Qué haríamos si quisiéramos que se ejecutaran unas ciertas órdenes en el caso de que la condición no se cumpliera? Sin duda podríamos añadir otro if que tuviera como condición la negación del primero:

```
if fav == "mundogeek.net":  
    print "Tienes buen gusto!"  
    print "Gracias"
```

```
if fav != "mundogeek.net":  
    print "Vaya, que lástima"
```

pero el condicional tiene una segunda construcción mucho más útil:

```
if fav == "mundogeek.net":  
    print "Tienes buen gusto!"  
    print "Gracias"  
else:  
    print "Vaya, que lástima"
```

if ... elif ... elif ... else

```
if numero < 0:  
    print "Negativo"  
elif numero > 0:  
    print "Positivo"  
else:  
    print "Cero"
```


if ... elif ... elif ... else

elif es una contracción de else if, por lo tanto *elif numero > 0* puede leerse como “si no, si numero es mayor que 0”. Es decir, primero se evalúa la condición del if. Si es cierta, se ejecuta su código y se continúa ejecutando el código posterior al condicional; si no se cumple, se evalúa la condición del elif. Si se cumple la condición del elif se ejecuta su código y se continua ejecutando el código posterior al condicional; si no se cumple y hay más de un elif se continúa con el siguiente en orden de aparición. Si no se cumple la condición del if ni de ninguno de los elif, se ejecuta el código del else.

Bucles

Mientras que los condicionales nos permiten ejecutar distintos fragmentos de código dependiendo de ciertas condiciones, los bucles nos permiten ejecutar un mismo fragmento de código un cierto número de veces, mientras se cumpla una determinada condición.

Sentencia while

El bucle while (mientras) ejecuta un fragmento de código mientras se cumpla una condición.

```
edad = 0
while edad < 18:
    edad = edad + 1
    print "Felicidades, tienes " + str(edad)
```

Por ejemplo, el siguiente programa escribe los números del 1 al diez:

```
i = 1
while i <= 11:
    print i,
    i = i + 1
```

contador.py

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

```
contador = 0
```

```
while (contador < 9):
```

```
    print 'El contador es:', contador
```

```
    contador = contador + 1
```

```
print "Good bye!"
```

Bubles infinitos

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

```
var = 1
```

```
while var == 1 :
```

```
    num = raw_input("Ingresa un numero :")
```

```
    print "Has ingresado: ", num
```

```
print "Alguna vez acabara \? "
```

Bucle: for ... in

En Python for se utiliza como una forma genérica de iterar sobre una secuencia. Y como tal intenta facilitar su uso para este fin.

Este es el aspecto de un bucle for en Python:

```
secuencia = ["uno", "dos", "tres"]  
for elemento in secuencia:  
    print elemento
```

El cuerpo del bucle se ejecuta tantas veces como elementos tenga la lista que utilices (o caracteres tenga la cadena que utilices). Por ejemplo, el programa

```
for i in [0,1,2,3,4] :  
    print "Hola",
```


Este programa también daría el mismo resultado

```
for i in "amigo" :  
    print "Hola",
```

La lista se puede a su vez generar con una función `range()`. El programa anterior se puede escribir así:

```
for i in range(5):  
    print "Hola",
```

Cambiando el argumento de la función range, puedes hacer que el programa salude muchas más veces.

```
for i in range(20):  
    print "Hola",
```

Un bucle for se limita a repetir el bloque de instrucciones, pero además, en cada iteración, la variable va tomando cada uno de los valores de la lista. Por ejemplo, el programa:

```
for i in range(5):  
    print "Hola. Ahora i vale",i,"y su cuadrado", i**2
```

La lista puede contener cualquier tipo de elementos, no sólo números. El bucle se repetirá siempre tantas veces como elementos tenga la lista y la variable irá tomando los valores de uno en uno. Por ejemplo, el programa:

```
for i in ['Alba', 'Benito', 'Carmen', 27]:  
    print "Hola. Ahora i vale",i,". Adios."
```

Este es un ejemplo de programa con acumulador:

```
suma = 0
for i in range(10):
    suma = suma + i
print "La suma de los números de 0 a 10 es", suma
```

Este es un ejemplo de programa con contador:

```
cuenta = 0
for i in range(1000):
    if i%7 == 0:
        cuenta = cuenta + 1
print "Entre 0 y 1000 hay", cuenta, "múltiplos de 7"
```

Bucles anidados

Un bucle anidado es un bucle situado en el cuerpo de otro bucle. Por ejemplo el programa:

```
for i in range(3):  
    for j in range(2):  
        print "i vale", i, "y j vale", j
```

Puedes utilizar la variable del bucle externo para controlar el bucle interno. Por ejemplo, el programa:

```
for i in range(4):  
    for j in range(i):  
        print "i vale", i, "y j vale", j
```

Funciones

Una función es un fragmento de código con un nombre asociado que realiza una serie de tareas y devuelve un valor. A los fragmentos de código que tienen un nombre asociado y no devuelven valores se les suele llamar procedimientos. En Python no existen los procedimientos, ya que cuando el programador no especifica un valor de retorno la función devuelve el valor None (nada), equivalente al null de Java.

En Python las funciones se declaran de la siguiente forma:

```
def mi_funcion(param1, param2):  
    print param1  
    print param2
```

Es decir, la palabra clave `def` seguida del nombre de la función y entre paréntesis los argumentos separados por comas. A continuación, en otra línea, indentado y después de los dos puntos tendríamos las líneas de código que conforman el código a ejecutar por la función.

También podemos encontrarnos con una cadena de texto como primera línea del cuerpo de la función. Estas cadenas se conocen con el nombre de docstring (cadena de documentación) y sirven, como su nombre indica, a modo de documentación de la función.

```
def mi_funcion(param1, param2):  
    """Esta funcion imprime los dos valores  
    pasados como parametros"""  
    print param1  
    print param2
```

Los valores por defecto para los parámetros se definen situando un signo igual después del nombre del parámetro y a continuación el valor por defecto:

```
def imprimir(texto, veces = 1):  
    print veces * texto
```

Para definir funciones con un número variable de argumentos colocamos un último parámetro para la función cuyo nombre debe precederse de un signo *:

```
def varios(param1, param2, *otros):  
    for val in otros:  
        print val
```

```
varios(1, 2)  
varios(1, 2, 3)  
varios(1, 2, 3, 4)
```

El manejo de archivos de texto

Desde el punto de vista de la programación un archivo no difiere en nada de los que utilizamos en un procesador de texto o en cualquier otra aplicación: simplemente lo abrimos, ejecutamos algún tipo de operación sobre él y luego lo volvemos a cerrar.

Archivos - Entrada y Salida

Veamos un ejemplo. Supongamos que existe un archivo llamado menu.txt que contiene una lista de comidas:

ensalada de tomate
papas fritas
pizza

Ahora vamos a escribir un programa que lea el archivo y nos muestre su contenido

```
# Primero abrimos el archivo en modo lectura (r)
inp = open("menu.txt","r")
# leemos el archivo, colocamos el contenido en una lista
# e imprimimos cada item
for linea in inp.readlines():
    print linea
# Ahora lo cerramos
inp.close()
```

Nota 1

`open()` requiere dos argumentos. El primero es el nombre del archivo (que puede ser pasado como una variable o directamente como una cadena de caracteres, como hicimos en el ejemplo). El segundo es el modo de acceso. El modo determina en qué forma se abrirá el archivo: para lectura (`r`) o para escritura (`w`). El modo también indica el tipo de contenido del archivo; así, agregando `"b"` a la `"r"` o `"w"` indicaremos que el archivo es binario (por ejemplo, `open(na,"rb")`); de lo contrario se da por supuesto de que se trata de un archivo en formato ASCII de texto.

Nota 2

Leemos y cerramos el archivo mediante funciones prefijadas con la variable de archivo. Esta notación se conoce como invocación de métodos y es nuestro primer vistazo a la Programación Orientada a Objetos.

Ejercicio

Consideremos ahora qué ocurre con los archivos más extensos. En primer lugar deberemos leer el archivo de a una línea por vez. También deberemos usar una variable `cuenta_lineas` que se incremente con cada línea leída; evaluaremos esta variable para ver si llegó a 25 (el número de líneas en la pantalla) y en ese caso pedirle al usuario que pulse una tecla ("Enter" por ejemplo) para continuar mostrando el listado, habiendo previamente puesto en cero la variable `cuenta_lineas`.

Realmente esto es todo. Abres un archivo, lees el contenido y lo manipulás en la forma que quieras. Cuando hayas terminado, sólo debés cerrar el archivo y a otra cosa.

Vamos a crear en Python nuestra propia versión del comando copy (copiar): simplemente abrimos un archivo en modo lectura, creamos uno nuevo en modo escritura y escribimos en este último las líneas del primero:

copiar.py

```
# Abrimos los archivos para lectura (r) y escritura (w)
inp = open("menu.txt","r")
outp = open("menu.bak","w")

# leemos el archivo, copiamos el contenido en una lista
# y copiamos ésta en el nuevo archivo
for linea in inp.readlines():
    outp.write(linea)

print "1 archivo copiado..."

# Ahora cerramos los archivos
inp.close()
outp.close()
```

Uso de funciones : Archivos de texto

Python cuenta con una clase llamada file que nos permite crear, escribir y leer datos de un archivo de texto.

Para crear un objeto de la clase file debemos utilizar la función open. Cuando llamamos a dicha función le pasamos como primer parámetro el nombre del archivo de texto y el modo de apertura del mismo:

```
open(nombre del archivo,modo)
```

Si el archivo de texto se encuentra en la misma carpeta que nuestro programa no necesitamos indicar el path (camino). Los modos de apertura del archivo de texto pueden ser:

- * 'r' Abre el archivo para lectura (debe existir el archivo)
- * 'w' Crea el archivo y lo abre para escribir
- * 'a' Abre el archivo para escribir. Se crea si el archivo no existe. Solo podemos agregar datos al final

Creación de un archivo de texto

El siguiente algoritmo crea en disco un archivo de texto llamado 'datos.txt' y no graba datos. Si queremos luego de ejecutar el programa podemos verificar la existencia del archivo en la misma carpeta donde almacenamos nuestro programa.

```
def creaciontxt():  
    archi=open('datos.txt','w')  
    archi.close()
```

```
creaciontxt()
```

Creamos una función llamada creaciontxt donde primero llamamos a la función open pasando como parámetros el nombre del archivo de texto a crear y el modo de apertura ('w')

La función open retorna la referencia del objeto file. Luego llamamos al método close de la clase file. Si luego queremos ver si se a creado el archivo de textos podemos hacerlo desde algun explorador de archivos, en la carpeta donde se encuentra nuestro programa en Python veremos un archivo llamado 'datos.txt' que tiene un tamaño de 0 bytes.

Grabación de líneas en el archivo de texto

```
def creartxt():  
    archi=open('datos.txt','w')  
    archi.close()
```

```
def grabartxt():  
    archi=open('datos.txt','a')  
    archi.write('Linea 1\n')  
    archi.write('Linea 2\n')  
    archi.write('Linea 3\n')  
    archi.close()
```

```
creartxt()  
grabartxt()
```

Lectura línea a línea de un archivo de texto

La clase file tiene un método llamado readline() que retorna toda una línea del archivo de texto y deja posicionado el puntero de archivo en la siguiente línea. Cuando llega al final del archivo readline retorna un string vacío.

```
def creartxt():  
    archi=open('datos.txt','w')  
    archi.close()
```

```
def grabartxt():  
    archi=open('datos.txt','a')  
    archi.write('Linea 1\n')  
    archi.write('Linea 2\n')  
    archi.write('Linea 3\n')  
    archi.close()
```

```
def leertxt():  
    archi=open('datos.txt','r')  
    linea=archi.readline()  
    while linea!="":  
        print linea  
        linea=archi.readline()  
    archi.close()
```

```
creartxt()  
grabartxt()  
leertxt()
```