Programación en Python --- Parte II - Ejemplos

Author: Patricio Páez Serrato
Date: 9 de Abril de 2004

Índice

1 Historia del documento

Este documento fué creado el 8 de Abril de 2004. Primera publicación el 27 de Junio de 2004. Segunda edición: 29 de Diciembre de 2008.

Para sugerencias,correcciones o preguntas acerca de este en documento, escribir a: nospam en pp punto com punto mx

El original de este documento se encuentra en http://pp.com.mx/python

2 Introducción

La primera parte de este documento es una referencia del lenguaje Python, con enlaces a páginas de interés (se encuentra en http://pp.com.mx/python). Esta segunda parte recopila los ejemplos que he mostrado cuando imparto el Tutorial de Python. Empiezan por algo muy sencillo, y combinando hasta llegar a algo más complejo. No por ser sencillos dejan de ser útiles, y como menciono durante el tutorial: los resultados de estos ejemplos pueden crecer y convertirse en sus propios conjuntos de herramientas.

2.1 Requisitos

Los ejemplos pueden realizarse tanto en una PC con MS-Windows, Unix o Linux siempre y cuando Python corra. Pueden usarse las versiones 1.5.2, y 2.0 en adelante. Los ejemplos tratan conceptos generales y procuran no usar características únicas en las versiones más recientes.

Todos los ejemplos, excepto las interfases gráficas pueden hacerse en modo texto, usando preferentemente un editor que resalte la sintaxis de Python como Vim, emacs, Scite y otros. Los usuarios de MS-Windows pueden instalar algunos de estos editores del CD Gnuwin.

Los ejemplos con Tkinter requieren tener el conjunto de funciones gráficas Tk así como el modulo de Python Tkinter que lo accesa. Ambos vienen incluídos en el instalador de Python para MS-Windows y se instalan cuando se aceptan todas las opciones del instalador sin modificación. En Linux el lector puede verificar mediante este comando:

```
[usuario@pc]$ rpm -qa | grep ^tk
tk-8.3.3-7mdk
tkinter-2.1.1-3mdk
```

2.2 Cómo realizar los ejemplos

Programar en Python puede hacerse de varias maneras según la necesidad o el gusto de cada persona. Para los neófitos mi recomendación es que utilicen el ambiente gráfico interactivo llamado *idle.* Esta herramienta viene incluída con el módulo tkinter. Además de resaltar la sintaxis en colores, permite editar archivos fuente y es más amigable al inicio.

El idle tiene dos ambientes: el shell interactivo con título **Python Shell** en su ventana; muestra el prompt >>> y espera un comando, y uno o más editores que se abren con el menú File --> New Window. Cada editor empieza con el título *Untitled* en su ventana, el cual cambia hasta que se salva a un archivo con File --> Save As (y subsecuentemente File --> Save). Cada editor nos permite ejecutar el código Python que contiene.

Se recomienda crear una carpeta para realizar y guardar los ejemplos. Para correr *idle*, cambiar primero a esa carpeta y entonces correr idle: En MS- Windows:

```
C:\ejemplos> C:\python22\idle\idle
```

En Linux:

```
[usuario@pc ejemplos]$ idle &
```

La primera vez que hacen un ejemplo, intenten hacerlo paso a paso en forma interactiva en el shell, tecleando cada comando. Es la forma en que aprenderán más que si simplemente copian y pegan.

Una vez que ya teclearon y funcionaron las cosas, entonces copien del shell interactivo y peguen a una ventana de editor y salven en un archivo con terminación .py para que conserven lo que hicieron para la posteridad.

3 Números grandes

Podemos explorar los enteros largos de Python si hacemos una tabla de potencias de 2, evaluando 2ˆn mientras n varía de 0 a n.

```
>>> for n in range(60):
        print n, 2**n
```

El resultado será:

```
0 1
1 2
2 4
3 8
4 16
5 32
6 64
7 128
8 256
9 512
10 1024
11 2048
12 4096
13 8192
14 16384
15 32768
16 65536
17 131072
18 262144
19 524288
20 1048576
21 2097152
22 4194304
23 8388608
24 16777216
25 33554432
26 67108864
27 134217728
28 268435456
29 536870912
30 1073741824
31 2147483648
32 4294967296
33 8589934592
34 17179869184
35 34359738368
36 68719476736
37 137438953472
38 274877906944
39 549755813888
40 1099511627776
41 2199023255552
42 4398046511104
43 8796093022208
```

```
44 17592186044416
45 35184372088832
46 70368744177664
47 140737488355328
48 281474976710656
49 562949953421312
50 1125899906842624
51 2251799813685248
52 4503599627370496
53 9007199254740992
54 18014398509481984
55 36028797018963968
56 72057594037927936
57 144115188075855872
58 288230376151711744
59 576460752303423488
```

Nota: desde la versión 2.2 de Python, si el resultado de una expresión entera excede el límite de los enteros (2^32 generalmente), éste es convertido automáticamente a entero largo. (mostrado con terminación L en el ambiente interactivo):

```
>>> 12345678901234
12345678901234L
```

En versiones 2.1 y anteriores tendremos un error:

```
>>> 12345678901234
OverflowError: integer literal too large
```

En el ejemplo anterior, lo que tendremos que hacer en este caso es agregar L al final del 2:

```
>>> for n in range(60):
        print n, 2L{*}{*}n
```

3.1 Agregando las comas

Los números muy grandes son difíciles de entender sin las comas que normalmente usamos. Vamos entonces a definir una función que agregue las comas. Los pasos necesarios serían:

- Convertir a cadena, para poder tomar cada dígito.
- Recorrer la cadena de derecha a izquierda, de tres en tres dígitos. Se inserta la coma cada vez si todavía quedan más de tres dígitos a la izquierda.

Al probar la función en el código anterior, remplazamos 2**n con poncoma(2**n) y además formamos columnas con el operador %:

```
def poncoma( n ):
    "Regresa n como cadena con comas."
    s = str(n)
    pos = len(s)
    while pos > 3:
        pos = pos - 3
        s = s[:pos] +  ',' + s[pos:]
    return s
```

```
for n in range(60):
    print '%3d %30s' % ( n, poncoma( 2**n ) )
```

Nota sobre versiones de Python: en las versiones 1.5.2 hasta 2.0, podemos utilizar la función *rjust()* del módulo *string*. Necesitaremos incluir el comando *import string* al principio del ejemplo*,* y referir la función como *string.rjust(str(n), 3)* y *string.rjust(poncoma(2**n), 30)*. A partir de las versión 2.1 todas las cadenas y las funciones que regresan cadena tienen la función *rjust()* incluída, y podríamos usar *str(n).rjust(3)* y *poncoma(2**n).rjust(30)*.

El resultado será ahora más legible:

```
 0                              1
 1                              2
 2                              4
 3                              8
 4                             16
 5                             32
 6                             64
 7                            128
 8                            256
 9                            512
10                          1,024
11                          2,048
12                          4,096
13                          8,192
14                         16,384
15                         32,768
16                         65,536
17                        131,072
18                        262,144
19                        524,288
20                      1,048,576
21                      2,097,152
22                      4,194,304
23                      8,388,608
24                     16,777,216
25                     33,554,432
26                     67,108,864
27                    134,217,728
28                    268,435,456
29                    536,870,912
30                  1,073,741,824
31                  2,147,483,648
32                  4,294,967,296
33                  8,589,934,592
34                 17,179,869,184
35                 34,359,738,368
36                 68,719,476,736
37                137,438,953,472
38                274,877,906,944
39                549,755,813,888
40              1,099,511,627,776
41              2,199,023,255,552
42              4,398,046,511,104
```

```
43              8,796,093,022,208
44             17,592,186,044,416
45             35,184,372,088,832
46             70,368,744,177,664
47            140,737,488,355,328
48            281,474,976,710,656
49            562,949,953,421,312
50          1,125,899,906,842,624
51          2,251,799,813,685,248
52          4,503,599,627,370,496
53          9,007,199,254,740,992
54         18,014,398,509,481,984
55         36,028,797,018,963,968
56         72,057,594,037,927,936
57        144,115,188,075,855,872
58        288,230,376,151,711,744
59        576,460,752,303,423,488
```

Le dejamos al lector el ejercicio de mejorar *poncoma()* añadiendo instrucciones para que acepte números con signo negativo, y con decimales.

4 Manejo de información

Hace años al Departamento de Informática se le llamaba de Procesamiento Electrónico de Datos (Electronic Data Processing en Inglés, con las siglas EDP). El nombre antiguo era más descriptivo de la labor que se supone debían realizar las computadoras. Aquí mostraremos cómo hacemos esto en Python, con un conjunto de datos pequeño pero real.

Tenemos la siguiente información en el archivo capitales.txt:

```
México,D.F.
Aguascalientes,Aguascalientes
Tijuana,Baja California
Mexicali,Baja California Sur
Campeche,Campeche
Tuxtla Gutiérrez,Chiapas
Chihuahua,Chihuahua
Saltillo,Coahuila
Colima,Colima
Durango,Durango
Toluca,Edo. de México
Chilpancingo,Guerrero
Guanajuato,Guanajuato
Pachuca,Hidalgo
Guadalajara,Jalisco
Morelia,Michoacán
Cuernavaca,Morelos
Tepic,Nayarit
Monterrey,Nuevo León
Oaxaca,Oaxaca
Puebla,Puebla
Querétaro,Querétaro
Chetumal,Quintana Roo
```

```
Culiacán,Sinaloa
Hermosillo,Sonora
San Luis Potosí,San Luis Potosí
Villa Hermosa,Tabasco
Tampico,Tamaulipas
Tlaxcala,Tlaxcala
Jalapa,Veracruz
Mérida,Yucatán
Zacatecas,Zacatecas
```

Son los nombres de los Estados mexicanos con su ciudad capital. Primero la capital, seguida de una coma y el nombre del Estado.

A continuación vamos a leer esta pequeña 'base de datos' a una cadena en Python, la vamos a convertir a una lista de registros, luego separaremos los campos de cada registro. El lector puede ir haciendo primero estos pasos en forma interactiva, revisando los resultados:

```
# Leemos el archivo a una cadena:

f = open( 'capitales.txt' )
datos = f.read()

# Partimos la cadena a una lista cuyos elementos son
# los registros de nuestra 'base de datos'

import string
lista = string.split( datos, '\n' )
print lista

# Convertimos cada elemento de la lista a una pareja,
# usamos una función lambda:

pares = map( lambda e: string.split( e, ','), lista )

print pares

# Podemos crear una lista para cada campo:
# una lista de capitales y otra lista de estados.
# Esta es una forma de obtenerlas:

capitales = []
estados = []
for capital,estado in pares:
    capitales.append( capital )
    estados.append( estado )

# Cómo determinar los nombres únicos de ambas listas?
# Usamos la concatenación de listas.

unicos = []
for nombre in capitales+estados:
    if nombre not in unicos:
        unicos.append(nombre)
    else:
        print nombre
```

```
# Cuántos elementos tenemos?

print len(unicos)

# Los ordenamos alfabéticamente:

unicos.sort()
print unicos
```

Para concluir, definimos una función que lee el archivo de este ejemplo, separa en campos y registros, y nos regresa una matriz en forma de una lista:

```
def leedatos( nombrearchivo ):
    "Abre un archivo, lee y regresa un arreglo."
    f = open( nombrearchivo )
    renglones = f.readlines()
    arreglo = map( lambda x: x.split(','), renglones )
    return arreglo

matriz = leedatos( 'capitales.txt' )

for renglon in matriz:
    for celda in renglon:
        print '%-20s' % celda,
    print
```

5 Generación de documentos HTML

Hoy en día usar HTML es cosa del diario, cuando navegamos en internet, leemos correo, y cada vez son más las aplicaciones que usan el navegador como punto de acceso al usuario. Muchas páginas de HTML se generan por programas, no por personas. Por esta razón vamos a practicar con Python.

Un documento básico de HTML puede ser así:

Unknown directive type "code-block".

```
.. code-block:: html

    <html>
    <head><title>página hecha a mano</title></head>
    <body>

    <h1>Hola...</h1>
    probando<br>

    </body>
    </html>
```

Si nunca habías hecho un documento HTML, puedes pegar el texto de arriba a un nuevo documento y salvarlo como estatica.html. Abre luego el documento con un navegador para que lo muestre.

5.1 Usando funciones

Vamos a dividir el documento HTML en las partes que lo forman, tomando como base la muestra anterior:

Unknown directive type "code-block".

```
.. code-block:: html

    ---------- inicio del documento  ----------
    <html>
    <head><title>página hecha a mano</title></head>
    <body>


    ---------- cuerpo del documento  ----------
    <h1>Hola...</h1>
    probando<br>



    ---------- final del documento  ----------
    </body>
    </html>
```

Ahora definiremos una función en Python para que regrese cada parte como texto. Empezamos con las funciones *inicio()* y *final()*. Luego agregamos para el cuerpo del documento las funciones *encabezado(), parrafo(), liga(), destino(), saltolinea() y horizontal()*.

Finalmente desarrollamos *tabla()* que será muy utilizada para mostrar arreglos. No es necesario utilizar variables i, j como en otros lenguajes para recorrer el arreglo, ni saber las dimensiones. Cada renglón del arreglo puede ser de distinta longitud.

Tenemos el siguiente código con las funciones básicas:

Unknown directive type "code-block".

```
.. code-block:: python

    def inicio( cadena='' ):
        "Inicia el documento HTML."
        return '''<html>
      <head>
        <title>''' + cadena + '''</title>
      </head>
    <body>\n'''

    def encabezado( nivel='1',  string='' ):
        "Regresa string con encabezado o título HTML, default es H1."
        return '<h'+ nivel + '>'+ string + '</h'+ nivel + '>'

    def horizontal():
        "Línea horizontal en el documento HTML."
        return '<hr />'

    def destino( clave, texto ):
        "Regresa texto como destino."
        return '<a name="' + clave + '>' + texto + '</a>'
```

```
    def liga( url, texto):
        "Liga o hipervínculo a url con texto."
        return '<a href="' + url + '">' + texto + '</a>'

    def parrafo( texto ):
        "Genera un párrafo con texto."
        return '<p>' + texto + '</p>\n'

    def saltolinea():
        "Genera un salto de línea."
        return '<br>\n'

    def tabla( arreglo ):
        """Muestra arreglo en una tabla HTML.

        Las celdas pueden ser cualquier tipo, se convierten a cadena siem-
pre."""

        temp =  '<table border="1">\n'

        for renglon in arreglo:
            temp = temp + '<tr>'
            for celda in renglon:
                temp = temp + '<td>' + str(celda) + '</td>\n'
            temp = temp + '</tr>\n'
        return temp + '</table>\n'

    def final():
        "Fin del documento HTML."
        return '''</body>
    </html>'''
```

Las funciones se van llamando en orden como se muestra en este ejemplo:

```
print inicio( 'Página dinamica')
print encabezado( 'Página generada con Python')
print parrafo( 'Tenemos a continuación una tabla:')
print tabla( [ ['r1c1', 'r1c2'],
               ['r2c1', 'r2c2'] ] )
print final()
```

Hemos pasado un arreglo en forma explícita, pero podemos usar una variable.
Las funciones que definimos pueden anidarse. Por ejemplo:

```
parrafo( 'El sitio ' + liga( 'http://www.python.org,'www.python.org')

+ 'tiene mucha información sobre Python.' )
```

para obtener:

El sitio www.python.org tiene mucha información sobre Python.

Para probar nuestros experimentos necesitamos enviar la salida del programa a un archivo que podemos llamar *prueba.**html*. Para hacer esto, necesitamos correr nuestro guión *html.py* con un comando así:

```
[user@host ejemplos]$ python html.py > prueba.html
```

En Windows:

```
C:\ejemplos> python html.py > prueba.html
```

Si Windows no tiene la ruta del ejecutable python.exe en su PATH, hay que agregarla. Generalmente es C:python22python:

```
C:\ejemplos> c:\python22\python html.py > prueba.html
```

La tabla podemos ahora construirla con una rutina, o llenarla con datos de alguna fuente, que será lo más usual. En el siguiente ejemplo hacemos ambas cosas. *hacertabla()* construye una matriz con expresiones. *leedatos()* es la rutina de la sección anterior sobre manejo de datos, que lee de nuestro archivo capitales.txt y usa lo que aprendimos para separar campos y registros. Aquí está el código:

```
def hacertabla( ren=10, col=10, operador='+'):
    """Construye y regresa un arreglo ren x col.

    Cada celda toma un valor que depende de x,y y el
    valor de operador.  El arreglo es de 10 x 10 si
    no se suministran los valores de ren y col."""

    array = []
    for ren in range(1, ren+1):
        array.append( [] )
        for col in range( 1, col+1):
            dato = calcula(ren, col, operador)
            array[-1].append( dato )
    return array


def calcula( a, b, operador):
    "Regresa una cadena tipo 'a operador b = resultado'."
    if operador == '+':
        return str(a) + ' + ' + str(b) + ' = ' + str( a+b)
    elif operador == '*':
        return str(a) + ' * ' + str(b) + ' = ' + str( a*b)

def leedatos( nombrearchivo ):
    "Abre un archivo, lee y regresa un arreglo."
    f = open( nombrearchivo )
    renglones = f.readlines()
    arreglo = map( lambda x: x.split(','), renglones )
    return arreglo

print inicio( 'Mi primera página dinamica')
print encabezado( 'Página generada con Python')
print parrafo( 'Algunos ejemplos de las rutinas para generar HTML')
print tabla( leedatos( 'capitales.txt' ) )
print encabezado( 'Tabla de sumas' )
print tabla ( hacertabla() )
print encabezado( 'Tabla de multiplicación' )
print parrafo( 'Esta tabla usa la misma función con otro contenido' )
print tabla ( hacertabla(12, 9, '*'), )
print final()
```

5.2 Usando una clase

El siguiente paso es definir una clase que llamaremos *buffer*. Esto representa mejoras respecto a nuestro ejemplo anterior, porque ahora podremos escribir el texto HTML a un archivo desde el programa sin redireccionar la salida estándar. Esto nos permite generar más de una página simultáneamente.

La clase *buffer* crea un atributo *text* en cada instancia que es el texto HTML que se va formando. El método *add()* se usa para agregar texto a la instancia, mientras que el método *pop()* es para regresar y vaciar el atributo de texto, como cuando lo vamos a escribir a un archivo.

La clase tiene también métodos con los mismos nombres que las funciones básicas. Estos métodos simplemente llaman a la función correspondiente y van agregando el texto regresado mediante el método *add()*.

```
class buffer:
    """Clase para generar objetos que almacenan codigo HTML
    que despues sera escrito a algun archivo."""

    def __init__( self ):
        "Constructor, crea el atributo de texto."
        self.text = ''

    def add( self, texto ):
        "Agrega texto arbitrario al buffer."
        self.text = self.text + texto

    def pop( self ):
        "Regresa el texto del buffer, y borra el buffer."
        temp = self.text
        self.text = ''
        return temp

    def inicio( self, titulo ):
        "Inicia el documento HTML."
        self.add( inicio( titulo ) )

    def encabezado( self, nivel, string):
        "Regresa string con encabezado o título HTML, default es H1."
        self.add( encabezado( nivel, string ) )

    def horizontal( self ):
        "Línea horizontal en el documento HTML."
        self.add( horizontal() )

    def destino( self, clave, texto ):
        "Regresa texto como destino."
        self.add( destino( clave, texto) )

    def liga( self, url, texto):
        "Liga o hipervínculo a url con texto."
        self.add( liga( url, texto) )

    def parrafo( self, texto ):
```

```
        "Genera un párrafo con texto."
        self.add( parrafo( texto) )

    def saltolinea( self ):
        "Genera un salto de línea."
        self.add( saltolinea() )

    def tabla( self, arreglo ):
        "Inserta arreglo en una tabla HTML."
        self.add( tabla( arreglo ) )

    def final( self ):
        "Fin del documento HTML."
        self.add( final() )
```

Ahora vemos un ejemplo de cómo usamos esta clase para crear páginas HTML. Creamos dos instancias de *buffer,* les agregamos contenido y finalmente escribimos cada buffer a un archivo. Cada página lleva un enlace a la otra:

```
import html

# Crear dos instancias de la clase buffer:

pagina1 = html.buffer()
pagina2 = html.buffer()

# Llenar cara buffer:

pagina1.inicio( 'Página 1' )
pagina1.encabezado( '2', 'Esta es página 1')
pagina1.liga( 'pag2.html', 'Ir a página 2' )
pagina1.final()

pagina2.inicio( 'Página 2' )
pagina2.encabezado( '2', 'Esta es Página 2')
pagina2.liga( 'pag1.html', 'Ir a página 1' )
pagina2.final()

# Vaciar cada buffer a un archivo:

fp = open( 'pag1.html', 'w' )
fp.write( pagina1.pop() )
fp = open( 'pag2.html', 'w' )
fp.write( pagina2.pop() )
```

6 Interfase gráfica de usuario

De los varios conjuntos de herramientas para hacer interfases gráficas, veremos Tk. Tk es el conjunto de herramientas que se hizo originalmente para el lenguaje de guiones TCL. Mediante el módulo Tkinter, Python puede utilizar Tk.

6.1 Introducción

Para trabajar con un cualquier juego de herramientas gráficas, necesitamos ir creando las ventanas o elementos (llamados también *widgets*) que usaremos, los cuales quedan en una jerarquía. En primer nivel tendremos una ventana principal, a la cual agregamos botones, un marco, un cuadro de texto, etc.

Los pasos para todo programa gráfico serán:

- Importar las clases y constantes del módulo Tkinter, mediante un comando import en su forma *import Tkinter.*
- Crear una instancia de la ventana raíz, a partir de la clase *Tkinter.Tk().*
- Crear una instancia de uno o más elementos para insertar en esta ventana raíz. Los elementos pueden ser widgets *Label, Canvas, Frame*, y otros. Un widget procesador de texto por ejemplo, se crea a partir de la clase *Tkinter.Text().*
- Enlazar funciones o métodos de la ventana raíz o alguno de los widgets a uno o más eventos del usuario. Los eventos son los movimientos y acciones del ratón y teclas presionadas en el teclado.
- Mostrar cada elemento con el método *pack()* que tiene definido.
- Agregar información a los elementos, como puede ser texto al editor o a una etiqueta
- Llamar el método *mainloop()* de nuestra ventana raíz, para que se active la atención a los eventos.
- Para terminar la aplicación se llama al método *quit()* de la ventana raíz.

Ya en código tendremos más o menos el siguiente esqueleto:

```
import Tkinter
raiz = Tkinter.Tk()
w = Tkinter.Label(raiz)
w.pack()
w[ 'text' ] = 'Hola mundo'
raiz.mainloop()
```

El ejemplo anterior se basa en el primer ejemplo de Fredrik Lundh en su documento *An introduction to Tkinter.*

6.2 Un editor

El widget de texto *Text()* nos da la base para hacer un editor. La clase Text() tiene los métodos *insert()* y *get()* para escribir o leer el contenido del buffer de texto que corresponde al área visible para el usuario. El código básico sería:

```
import Tkinter, sys

raiz = Tkinter.Tk()
editor = Tkinter.Text(raiz)
editor.pack()

f = open( sys.argv[1] )
editor.insert( '1.0', f.read() )

raiz.mainloop()
```

Al código anterior le agregamos instrucciones en el bloque principal indicar que se requiere el nombre del archivo. El bloque maneja la excepción en caso de que el archivo no exista y se trate de un archivo nuevo a crear. Se agregan también las funciones *quit()* para terminar la aplicación, y *save()* para guardar el contenido del buffer en el archivo. La aplicación se termina mediante la combinación de teclas Control+q y el archivo se salva con las teclas Control+s. Estos dos eventos de teclado se llaman *<Control-q>* y *<Control-s>* en Tkinter*,* y se ligan las funciones *quit()* y *save()* a estos eventos mediante el método *bind* que tienen todos los widgets en Tk:

```
import Tkinter, sys

def quit(event ):
    "Termina la aplicación."
    raiz.quit()

def save(event):
    "Salva el texto del editor."
    f = open( nombrearchivo, 'w' )
    f.write( editor.get( '1.0', END) )
    f.close()
    print 'archivo', nombrearchivo, 'salvado.'

raiz = Tkinter.Tk()
editor = Tkinter.Text(raiz)
editor.pack()
editor.bind( '<Control-q>', quit )
editor.bind( '<Control-s>', save )

if len(sys.argv)>1:
    nombrearchivo = sys.argv[1]
    try:
        f = open( nombrearchivo )
        editor.insert( '1.0', f.read() )
    except:
        print nombrearchivo, 'es archivo nuevo'
else:
    print 'falta proporcionar nombre de archivo'
    sys.exit()

mainloop()
```

Este primer ejemplo nos muestra lo fácil que es crear una aplicación gráfica directamente. Para casos de interfases más complejas, existen herramientas como Glade que ayudan a crearla en forma gráfica.

6.3 Usando una clase

El caso anterior tiene una sola ventana de texto. Para tener varios archivos abiertos no tenemos que repetir el código para cada ventana. Lo que utilizamos es una clase y cada ventana es una instancia de la misma.

El código base para crear un editor que ya vimos será incluído en una clase *editor*, específicamente en la función *__init__()* que es su función contructora y se ejecuta cuando se crea una instancia.

Tk siempre crea una instancia de la clase *Tk()*, una ventana principal, que podría ser la ventana de inicio de la aplicación. En este caso la ocultamos desde el inicio con el método *withdraw()*.

La clase editor tiene el método *quit()* para terminar la aplicación, y *siguiente()* y *anterior()* que son para activar otra de las ventanas de edición utilizando las teclas *página siguiente* y *página anterior* del teclado, cuyos eventos en Tkinter se llaman *<Prior>* y *<Next>*.

Usamos una lista *editores* inicialmente vacía, mediante un ciclo se va agregando una instancia de editor para cada parámetro que se proporcionó. El código es así:

```
class editor:
    "Un area de texto simple."

    def __init__( self, nombrearchivo ):
        "Construye la instancia."
        self.tl = Tkinter.Toplevel()
        self.tl.title (nombrearchivo)
        self.nombrearchivo = nombrearchivo
        self.buffer = Tkinter.Text( self.tl, height=10)
        self.buffer.pack()
        self.buffer[ 'wrap'] = Tkinter.NONE
        self.buffer[ 'background'] = 'white'
        self.buffer[ 'foreground' ] = 'black'
        self.buffer[ 'font'] = ( 'verdana', 14 )
        self.buffer.bind( '<Prior>', self.anterior )
        self.buffer.bind( '<Next>', self.siguiente)
        self.buffer.bind( '<Control-q>', self.quit)

        self.s = string.replace( open( self.nombrearchivo ).read(),'\015', '')
        self.buffer.insert( Tkinter.END, self.s )

        if d.has_key( self.nombrearchivo ):
            self.tl.geometry( d[ self.nombrearchivo ] )
        lineas = self.s.splitlines()
        self.buffer['height'] = len( lineas ) + 1
        self.buffer['width'] = max( map(len, lineas) ) + 1

    def anterior( self, event ):
        i = notas.index( self) - 1
        if i < 0:  i = len(notas)-1
        notas[i].buffer.focus_set()
        notas[i].tl.lift()
        return 'break'

    def siguiente( self, event ):
        i = ( notas.index( self) + 1 ) % len( notas )
        notas[i].buffer.focus_set()
        notas[i].tl.lift()
        return 'break'

    def quit( self, event ):
        "Actualiza posiciones, cierra ventanas, termina."
        f = open( archivoconf, 'w' )
        for nota in notas:
            d[ nota.nombrearchivo ] = nota.tl.geometry()
            nota.tl.withdraw()
        for llave, dato in sorted( d.items() ):
```

```
            f.write( llave + ' ' + dato + '\n')
            nota.tl.withdraw()
        root.quit()

def LeeConf():
    "Lee nombres y posiciones de archivoconf."
    lineas = []
    try:
        f = open( archivoconf )
        lineas = map( string.split, f.read().splitlines() )
    except:
        print archivoconf, 'no existe'
    for llave, dato in lineas:
        d[llave] = dato


import sys, string, Tkinter

archivoconf = 'editor.conf'
nombresarchivos = sys.argv[ 1:]
if nombresarchivos:
    d = {}
    root = Tkinter.Tk()
    root.withdraw()

    LeeConf()
    notas = []
    for nombrearchivo in nombresarchivos:
        notas.append( editor( nombrearchivo ) )
    notas[0].buffer.focus_set()

    root.mainloop()
```

7 Servidor HTTP

Python nos permite hacer aplicaciones que incluyan un servidor de HTTP. Puede ser en la intranet de nuestra casa u oficina, o bien en internet. Un servidor básico de HTTP, al cual podemos modificar si deseamos, empieza así:

```
import BaseHTTPServer, SimpleHTTPServer
httpd = BaseHTTPServer.HTTPServer( ( '', 80),
                        SimpleHTTPServer.SimpleHTTPRequestHandler)
httpd.serve_forever()
```

Para añadir CGI utilizamos *CGIHTTPRequestHandler* en lugar de *SimpleHTTPRequestHandler* en nuestra instancia de servidor:

```
import BaseHTTPServer, CGIHTTPServer
httpd = BaseHTTPServer.HTTPServer( ( '', 80),
                        CGIHTTPServer.CGIHTTPRequestHandler)
httpd.serve_forever()
```

Ambos casos pueden probarse en tu máquina tratando de mostrar la dirección *http://localhost* en un navegador. Podrás ver los archivos que estén en el directorio de trabajo al momento de correr el guión servidor. Mientras el guión corre, arroja una bitácora de los accesos. En Linux es necesario que estés como root para correr estos guiones.

Roundup es un ejemplo de una aplicación hecha en Python que crea su servidor HTTP con CGI. Ver http://roundup.sf.net para más información.

8 Toques finales

8.1 Documentación

Documentar nuestros programas, módulos y ejemplos en Python no representa un esfuerzo adicional si hemos incluído las cadenas de documentación. Para el caso de un módulo, podemos incluir las siguientes cadenas:

```
'''Cadena de descripción del módulo

Puede tener una línea de encabezado,
y después uno o más párrafos de explicación...'''

__author__ = 'xxxx xxxx'
__date__ = 'xxxx xxxx'
__version__ = 'xxxx xxxx'
__credits__ = 'xxxx xxxx'
__text__ = 'xxxx xxxx'
__file__ = 'xxxx xxxx'
```

Las funciones para generar HTML las incluímos todas en un archivo *html.py*, al cual agregamos al inicio las cadenas anteriores. Después con la herramienta *pydoc* incluída con Python, podemos generar un archivo *html.**html* con la documentación del mismo:

```
[usuario@pc ejemplos]$ /usr/lib/python2.1/pydoc.py -w ./html.py
```

Escribimos './' antes del nombre del archivo, para que pydoc sepa que es un archivo y no un módulo predefinido.

8.2 Prueba de módulos

Se recomienda al final de un módulo definir una función *prueba()* que llame algunas de las funciones para validar que todo está bien. Esta función deberá correr solamente cuando el módulo se carga como guión principal, no cuando se importa desde otro archivo. Para lograr esto, usar el siguiente código:

```
if __name__ == '__main__':
    prueba()
```