Un Caso de Estudio

Para este caso de estudio nos abocaremos a ampliar el contador de palabras que desarrollamos en un capítulo anterior. Crearemos un programa que imita al wc de Unix, en tanto muestra el número de renglones, palabras y caracteres de un archivo. Sin embargo, nosotros iremos un poco más allá y contaremos también la cantidad de oraciones, frases, palabras, letras y signos de puntuación en un archivo de texto. Seguiremos el desarrollo de este programa en etapas, incrementando gradualmente sus capacidades. Luego lo convertiremos en un módulo para ser reutilizado, y finalmente le aplicaremos técnicas de Programación Orientada a Objetos.

Implementaremos el programa en Python, pero las fases iniciales también podrían codificarse en BASIC o Tcl. Cuando avancemos a las etapas más complejas deberemos manejarnos con estructuras de datos para lo cual Python nos resultará mejor, aunque Tcl sigue siendo una opción aceptable. Finalmente, para las técnicas de POO sólo podremos usar Python.

Podríamos implementar también una serie de características adicionales, que dejo en manos del lector curioso:

- calcular el índice FOG del texto,
- calcular el número de palabras únicas y su frecuencia,
- crear una nueva versión que analice archivos RTF

Contando renglones, palabras y caracteres

Revisemos nuestro contador de palabras anterior:

```
import string
def numpalabras(s):
    lista = string.split(s)
    return len(lista)

inp = open("menu.txt","r")
total = 0

# acumulamos los totales de cada línea
for linea in inp.readlines():
    total = total + numpalabras(linea)
print "El archivo tiene %d palabras" % total

inp.close()
```

Ahora necesitamos agregar un contador de líneas y de letras. El contador de renglones es sencillo ya que sólo tenemos que incrementar una variable en cada iteración del bucle que lee las líneas. El contador de letras es también sencillo: deberemos iterar sobre la lista de palabras agregando en una variable la cantidad de letras de cada una de ellas.

También tenemos que lograr que nuestro programa sirva a un propósito más general, por ejemplo, leyendo el nombre del archivo de la línea de comando y requiriéndoselo al usuario si no lo ha provisto en ella. (Una estrategia alternativa sería leer de la pantalla estándar como es el caso del wc real.)

Así quedaría nuestro propio wc:

```
import sys, string

# Obtener el nombre del archivo de la línea de comando o del usuario
if len(sys.argv) < 2:
    nombre = raw_input("¿Cuál es el nombre del archivo: ")
else:
    nombre = sys.argv[1]

inp = open(nombre,"r")

# inicializamos los contadores en cero, lo cual también crea las variables
palabras = 0
renglones = 0
letras = 0

for linea in inp.readlines():
    renglones = renglones + 1

    # la separamos en una lista de palabras y las contamos
    lista = string.split(linea)
    palabras = palabras + len(lista)

    # agregamos las letras de las palabras y los espacios que perdimos
    # al separar la línea
    for palabra in lista:
       letras = letras + len(palabra)
    letras = letras + len(lista)
```

```
print "%s tiene %d renglones, %d palabras y %d letras" % (nombre, renglones, palabras, letras)
inp.close()
```

Si utilizás el comando wc de Unix, sabrás que éste acepta un nombre de archivo con comodines para mostrar los resultados de todos los archivos que cumplan la condición de los comodines, así como también la posibilidad de obtener el total general. Nuestro programa sólo acepta nombres de archivo comunes. Si querés ampliar sus posibilidades e incluir la opción del uso de los comodines en el nombre de archivo, revisá el modulo glob y creá una lista con los nombres sobre la cual iterar cada vez. Será necesario utilizar contadores temporarios para cada archivo y luego contadores acumulativos para el total general. Otra posibilidad sería usar un diccionario...

Contando oraciones en vez de renglones

Cuando comencé a pensar en cómo podíamos extender las capacidades de nuestro programa para que pudiera contar oraciones y palabras (y no "grupos de caracteres" como hemos hecho antes), mi primera idea fue extraer en una lista cada línea del archivo, luego crear una nueva lista con cada palabra en cada línea, y finalmente procesar cada "palabra" quitándole los caracteres extras. Sin embargo, este procedimiento es muy tedioso dado que en la práctica deberemos iterar tres veces sobre el archivo, lo cual funciona bien pero es muy lento.

Resulta entonces evidente que si extraemos las palabras y los signos de puntuación, podremos utilizar estos últimos para contar oraciones, frases, etc. (definiendo una oración o una frase en términos de puntuación).

Este avance nos permite iterar una sola vez sobre el archivo, y otra vez sobre la lista de puntuación (que es mucho más breve). Veamos cómo codificar esta modificación en pseudo-código:

```
para cada linea en archivo:
   incrementar contadorlinea
   si linea vacía:
       incrementar contadorpárrafo
   dividir línea en grupos de caracteres

para cada grupo de caracteres:
   incrementar contadorgrupo
   extraer puntuación en un diccionario - {char:count}
   si no quedan caracteres:
      borrar grupo
   de lo contrario: incrementar contador palabra

contador oraciones = suma de('.', '?', '!')
contador de frases = suma de toda la puntuación (una definición muy pobre...)

mostrar párrafos, líneas, oraciones, frases, grupos, palabras
para cada signo puntuación:
  mostrar contador count
```

Visto de esta forma, podríamos crear cuatro funciones usando la agrupación que hemos puesto más arriba. Esto nos ayudará a escribir un módulo que pueda ser reutilizado completamente o al menos de manera parcial.

Convirtiéndolo en un módulo

Las funciones principales son: `obtenerGruposCaracteres(archivo)`, y `obtenerPuntuacion(listaPalabras)`. Veamos en qué se transforma nuestro pseudo- código...

```
############################
# Módulo: grammar
# Creado: A.J. Gauld, 1999,8,19
#
# Funcionamiento:
# cuenta párrafos, líneas, oraciones, "frases", grupos de caracteres,
# palabras y signos de puntuación de un archivo de texto. Asume que las
# oraciones terminan en los signos [.!?] y que los párrafos tienen un renglón
# en blanco antes y después. Una "frase" es simplemente un segmento de una oración
# separado por puntuación (no es muy original, quizás algún día lo mejoremos)
#
# Uso: En su uso básico, toma un nombre de archivo y devuelve la estadística de
#      dicho archivo. Se espera que un segundo módulo utilice estas funciones
#      para producir comandos más utiles.
############################
import string, sys

############################
# inicializar las variables globales
c_parrafo = 1 # asumimos al menos 1 párrafo!
c_renglon, c_oracion, c_frase, c_palabra = 0,0,0,0
puntuacion = {}
```

```
grupos = []
alphas = string.letters + string.digits
limites = ['.','?','!']
signos_puntuacion = ['&','(',')','-',';',':',','] + limites
c_puntuacion = {}
for c in signos_puntuacion:
    c_puntuacion[c] = 0
formato = """%s contiene:
%d párrafos, %d renglones y %d oraciones.
Estas a su vez contienen %d frases y un total de %d palabras."""


############################
# Ahora definimos las funciones que harán el trabajo

def obtenerGruposCaracteres(archivo):
    pass

def obtenerPuntuacion(listaPalabras):
    pass

def mostrarResultados():
    print format % (sys.argv[1],
                    c_parrafo, c_renglon, c_oracion,
                    c_frase, c_palabra)

def Analizar(archivo):
    global grupos  # usamos la global
    grupos = obtenerGruposCaracteres(archivo)
    obtenerPuntuacion(grupos)
    mostrarResultados()


# Hacemos que se ejecute si es llamado desde la línea de comando (en este caso
# la variable mágica __name__ se convierte en '__main__'

if __name__ == "__main__":
     if len(sys.argv) <> 2:
        print "Uso: python grammar.py < nombre de archivo >"
        sys.exit()
     else:
        Documento = open(sys.argv[1],"r")
        Analizar(Documento)
```

En lugar de mostrar todo en un largo listado, prefiero discutir este esqueleto básico, para luego analizar cada una de las tres funciones principales. Para que el programa funcione, deberás "pegarlo" todo al final.

Lo primero que quisiera que notes es el comentario al principio del programa. Esta es una práctica común que permite a los usuarios del archivo saber qué contiene y cómo debe utilizarse. La información de la versión (autor y fecha) es útil ya que nos permite comparar los resultados con otra persona que tenga una versión más actualizada.

La sección final es una característica de Python, que llama "__main__" a cualquier módulo cargado desde la línea de comando. Podemos evaluar esta variable especial llamada __name__ , y si su contenido es "__main__" sabremos que el módulo no ha sido importado, sino que ha sido cargado directamente desde la línea de comando, por lo cual ejecutamos nuestra condición `if`.

Este código condicional incluye una advertencia amistosa para el usuario que impide que el programa se ejecute si no se ha provisto el nombre del archivo (o eventualmente si se han colocado demasiados archivos).

Notemos finalmente que la función `Analizar()` simplemente llama a las demás funciones en el orden correcto. Esta también es una práctica común que permite al usuario utilizar toda la funcionalidad básica del programa de una manera sencilla (con `Analizar()`), o si no, realizar llamadas individuales a las funciones primitivas de bajo nivel.

obtenerGruposCaracteres()

El pseudo-código para este segmento era:

```
para cada línea en archivo:
   incrementar contador de líneas
   si línea vacía:
       incrementar contador de párrafos
   dividir la línea en grupos de caracteres
```

Podemos implementar esto en Python de manera muy sencilla:

```
# usamos los contadores globales y la lista de grupos de caracteres
def obtenerGruposCaracteres(archivo):
   global c_parrafo, c_renglon, grupos
   try:
      for linea in archivo.readlines():
         c_renglon = c_renglon + 1
         if len(linea) == 1: # es el caso de un renglón en blanco (corte de párrafo)
             c_parrafo = c_parrafo + 1
         else:
             grupos = grupos + string.split(linea)
   except:
       print "Imposible leer el archivo"
       sys.exit()
```

Nota 1: Debemos utilizar la instrucción `global` para declarar variables que ya han sido creadas afuera de la función. De no usar esta instrucción, Python crearía dentro de la función nuevas variables locales con el mismo nombre (en este caso, cambiar el valor local no tendría ningún efecto en las variables globales del mismo nombre).

Nota 2: Hemos utilizado una cláusula `try/except` para detectar cualquier tipo de error y detener el programa.

obtenerPuntuacion()

Para esta función requeriremos de un mayor esfuerzo y la utilización de una serie de características nuevas de Python.

Nuestro pseudo-código era:

```
para cada grupo de caracteres:
   incrementar contador de grupos
   extraer signos de puntuación en un diccionario - {char:count}
   si no hay más caracteres:
      borrar grupo
   de lo contrario: incrementar contador palabras
```

Mi primer intento es el que sigue:

```
def obtenerPuntuacion(listaPalabras):
    global puntuacion
    for item in listaPalabras:
        while item and (item[-1] not in alphas):
            p = item[-1]
            item = item[:-1]
            if p in puntuacion.keys():
                puntuacion[p] = puntuacion[p] + 1
            else: puntuacion[p] = 1
```

Fijate que esta versión no incluye la clásula final `if/else` que habíamos puesto en el pseudo-código. La eliminé para simplificar un poco las cosas y porque me parece que en la práctica nos encontraríamos con muy pocas palabras formadas únicamente por signos de puntuación. Sin embargo, la añadiremos a la versión definitiva de nuestro programa.

Nota 1: La lista de palabras se ha convertido en un parámetro, de modo que los usuarios del módulo puedan incluir sus propias listas y no verse forzados a trabajar desde un archivo.

Nota 2: Asignamos `item[:-1]` a `item`. Esta característica se denomina *slicing* (tajadas) en Python. Los dos puntos (:) indican que el índice debe ser tratado como un rango. Podríamos haber especificado `item[3:6]` para extraer en una lista `item[3}, item[4]` e `item[5]`.

El rango por defecto es el comienzo o el final de la lista dependiendo de qué costado de los dos puntos quedan en blanco. Así, `item[3:]` indica todos los miembros de item desde el [3] hasta el final. Esta es realmente una función muy util de Python. La lista original `item` desaparece y la nueva lista se asigna a `item`.

Nota 3: Utilizamos un índice negativo para extraer el último caracter de `item`. También realizamos un bucle por si hay múltiples signos de puntuación al final de un grupo.

Al probar el programa, nos daremos cuenta de que necesitamos hacer lo mismo al comienzo del grupo, ya que podemos detectar el cierre de paréntesis pero no la apertura de los mismos. Para solucionar esto crearemos una nueva función `trim()` que removerá la puntuación al comienzo y al final de un grupo:

```
def trim(item):
    global c_puntuacion
    # quitamos del comienzo
    while (len(item) > 0) and (item[0] not in alphas):
         ch = item[0]
         if ch in c_puntuacion.keys():
```

```
            c_puntuacion[ch] = c_puntuacion[ch] + 1
        item = item[1:]
    # ahora de atrás
    while (len(item) > 0) and (item[-1] not in alphas):
        ch = item[-1]
        if ch in c_puntuacion.keys():
            c_puntuacion[ch] = c_puntuacion[ch] + 1
        item = item[:-1]
    return item
```

Entonces `obtenerPuntuacion()` se vuelve un tanto trivial:

```
def obtenerPuntuacion(listaPalabras):
    # quitar la puntuación de un grupo de caracteres
    for item in listaPalabras:
        item = trim(item)
    # ahora borramos las "palabras" vacías
    for i in range(len(listaPalabras)):
        if len(listaPalabras[i]) == 0:
            del(listaPalabras[i])
```

Nota 1: Ahora hemos incluido la eliminación de palabras "vacías".

Note 2: La función trim contiene dos bloques de código prácticamente idéntico. Esto es con seguridad un error de programación que deberemos corregir rediseñando el código. Desafortunadamente, no se me ocurre una solución por el momento, pero bueh... esto es la programación...

Note 3: En función de la reutilización de nuestra función, deberíamos haber partido nuestra función trim en partes aún más pequeñas. Esto nos habría permitido crear una función para remover un signo de puntuación tanto del comienzo como del final de una palabra y devolviendo precisamente el signo eliminado. Luego, una nueva función llamaría repetidas veces a nuestra función trim para obtener el resultado buscado. Desde el momento en que el objetivo de nuestro módulo es producir una estadística del texto y no un procesamiento general del mismo, si quisieramos introducir esta modificación de una manera correcta deberíamos crear un módulo que luego importaremos. Sin embargo, dado que este módulo sólo contendría una única función, no creo que sea demasiado útil. Por esta razón lo dejaremos tal cual como está ahora.

El módulo grammar final

Lo único que nos falta es mejorar la presentación de los resultados para que esta incluya los signos de puntuación y los contadores. Reemplazaremos la función `mostrarResultados()` anterior con la siguiente:

```
def mostrarResultados():
    global c_oracion, c_frase
    for p in limites:
        c_oracion = c_oracion + c_puntuacion[p]
    for c in c_puntuacion.keys():
        c_frase = c_frase + c_puntuacion[c]
    print format % (sys.argv[1],
                    c_parrafo, c_renglon, c_oracion,
                    c_frase, len(grupos))
    print "Se han utilizado los siguientes signos de puntuación:"
    for p in c_puntuacion.keys():
        print "\t%s\t:\t%3d" % (p, c_puntuacion[p])
```

Si has logrado reacomodar todo lo anterior en su lugar ahora podrías escribir en la línea de comandos:

```
C:>python grammar.py miarchivo.txt
```

y obener los resultados estadísticos del archivo *miarchivo.txt* (o como este se llame). Es materia opinable cuán útil puede resultarte este programa, pero no podemos negar que a través de la evolución de nuestro simple programita has obtenido algunas ideas de cómo crear tus propios programas. Lo más importante es probar una y otra vez las cosas, agregando, quitando y modificando todo aquello que no parezca correcto (o directamente no funcione como esperabas). No debés avergonzarte de probar diversos caminos, la mayor parte de las veces uno aprende mucho en el proceso.

Para concluir nuestro curso reescribiremos el módulo grammar utilizando técnicas de la Programación Orientada a Objetos (POO). En el proceso de conversión verás cómo un abordaje de POO produce módulos mucho más flexibles para el usuario y más fácilmente extensibles.

Clases y objetos

Uno de los mayores problemas que encontrará el usuario de nuestro módulo es la confianza en las variables globales. Estos significa que el módulo es capaz de analizar únicamente un documento a la vez: cualquier intento por manejar más de un texto producirá que las variables globales sean sobreescritas.

Por el contrario, al colocar estos valores globales dentro de una clase podemos crear múltiples instancias de la clase (una por cada archivo) y cada una de ella contiene sus propias variables. Además, si nuestros métodos se reconstruyen de una manera lo suficientemente

granular, podremos crear una arquitectura por medio de la cual sean fácilmente modificables los criterios de búsqueda y de organización para agregar a los archivos de texto otros tipos de documentos (por ejemplo, palabras en un documento HTML, etc.).

Nuestro primer intento es este:

```
#! /usr/local/bin/python
################################
# Modulo: documento.py
# Autor: A.J. Gauld
# Fecha:   199/08/22
# Versión: 0.1
################################
# Este módulo provee la clase Documento
# que puede ser subclasada para diferentes tipos
# de Documentos (texto, HTML, Latex, etc). Presentamos
# como ejemplos el caso de Texto y el HTML.
#
# Los servicios primarios incluyen
#    - obtenerGruposCaracteres(),
#    - obtenerPalabras(),
#    - mostrarResultados().
################################
import sys,string

class Documento:
    def __init__(self, archivo):
        self.archivo = archivo
        self.infile = open(archivo,"r")
        self.c_parrafo = 1
        self.c_renglon, self.c_oracion, self.c_frase, self.c_palabras = 0,0,0,0
        self.alphas = string.letters + string.digits
        self.limites = ['.','?','!']
        self.signos_puntuacion = ['&','(',')','-',';',':',','] + stop_tokens
        self.c_puntuacion = {}
        self.grupos = []
        for c in self.signos_puntuacion + self.limites:
            self.c_puntuacion[c] = 0
        self.formato = """%s contiene:
%d párrafos, %d renglones y %d oraciones.
Estas a su vez contienen %d frases y un total de %d palabras."""

    def obtenerGruposCaracteres(self):
        try:
           f = self.infile
           for linea in f.readlines():
              self.c_renglon = self.c_renglon + 1
              if len(linea) == 1: # línea en blanco (corte de párrafo)
                 self.c_parrafo = self.c_parrafo + 1
              else:
                 self.grupos = self.grupos + string.split(linea)
        except:
              print "No se pudo leer el archivo", self.archivo
              sys.exit()

    def obtenerPalabras(self):
        pass

    def mostrarResultados(self, parr=1, renglones=1, oraciones=1, palabras=1, puntos=1):
        pass

    def Analizar(self):
        self.obtenerGruposCaracteres()
        self.obtenerPalabras()
        self.mostrarResultados()

class DocumentoTexto(Documento):
        pass

class DocumentoHTML(Documento):
        pass

if __name__ == "__main__":
     if len(sys.argv) <> 2:
        print "Uso: python documento.py "
        sys.exit()
     else:
        D = Documento(sys.argv[1])
        D.Analizar()
```

Para implementar la clase debemos definir el método `obtenerPalabras` Podemos simplemente copiar lo que hicimos en la versión anterior y crear un método "trim". Sin embargo, deseamos que en esta versión que utiliza la POO nuestro método sea fácilmente ampliable y extendible, por lo cual dividiremos a "obtenerPalabras" en una serie de pasos. Luego en las subclases sólo deberemos dejar de lado estos pasos y no reescribir todo el método. Esto nos permitirá una mayor versatilidad para manejarnos con diferentes tipos de documentos.

De manera específica agregaremos métodos que rechacen grupos que nosotros reconocemos como inválidos y quiten los caracteres innecesarios del comienzo y del final de una oración. Entonces, agregamos estos tres métodos a la clase `Documento` e implementamos `obtenerPalabras` a partir de estos métodos.

```
class Documento:
   # .... igual que antes
   def obtenerPalabras(self):
       for w in self.grupos:
           self.ltrim(w)
           self.rtrim(w)
       self.removerExcepciones()

   def removerExcepciones(self):
       pass

   def ltrim(self,palabra):
       pass

   def rtrim(self,palabra):
       pass
```

Habrás notado que hemos definido el cuerpo de los métodos utilizando la instrucción `pass`, la cual no hace absolutamente nada. En su lugar, definiremos cómo estos métodos operan con cada tipo de documento en particular.

Documentos de Texto

Para un documento de texto codificaremos:

```
class DocumentoTexto(Documento):
     def ltrim(self,palabra):
         while (len(palabra) > 0) and (palabra[0] not in self.alphas):
             ch = palabra[0]
             if ch in self.c_puntuacion.keys():
                  self.c_puntuacion[ch] = self.c_puntuacion[ch] + 1
             palabra = palabra[1:]
         return palabra

     def rtrim(self,palabra):
         while (len(palabra) > 0) and (palabra[-1] not in self.alphas):
             ch = palabra[-1]
             if ch in self.c_puntuacion.keys():
                  self.c_puntuacion[ch] = self.c_puntuacion[ch] + 1
             palabra = palabra[:-1]

     def removerExcepciones(self):
         top = len(self.grupos)
         i = 0
         while i < top:
             if (len(self.grupos[i]) == 0):
                del(self.grupos[i])
                top = top - 1
             i = i+1
```

Las nuevas funciones "trim" son virtualmente idénticas a la que habíamos creado en el módulo `grammar.py`, pero esta vez las hemos partido en dos. La función `removerExcepciones` ha sido definida para borrar las palabras vacías.

Fijáte que he cambiado la estructura del último módulo al agregar un bucle `while` en lugar del `for` anterior. La modificación se debe a que en la fase de prueba nos dimos cuenta de que si borrábamos elementos de la lista, el `rango` (calculado al comienzo) mantenía todavía el largo original y esto producía que intentáramos acceder a miembros de la lista inexistentes. Para evitar este error utilizamos un bucle `while` y ajustamos el índice máximo cada vez que eliminamos un elemento.

Documentos HTML

Respecto del HTML partiremos de una visión un tanto simplista y nos contentaremos con remover los grupos de caracteres que comiencen o terminen con '<' o '>'. Haremos esto antes de quitar la puntuación con los métodos `ltrim` y `rtrim`. Esto significa que deberemos redefinir `obtenerPalabras`. Para quitar la puntuación usaremos los mismos métodos que para los documentos de texto, por lo cual, en vez de heredar directamente de la clase `Documento`, heredaremos, en cambio, de `DocumentoTexto` y reutilizaremos los métodos `trim`.

Entonces, nuestra clase DocumentoHTML sería así:

```
class DocumentoHTML(DocumentoTexto):
      def removerExcepciones(self):
         top = len(self.grupos)
         i = 0
         while i < top:
             if (self.grupos[i][0] == '<') or \
                (self.grupos[i][-1] == '>'):
                del(self.grupos[i])
                top = top - 1
             i = i+1

      def obtenerPalabras(self):
          self.removerExcepciones()
          for i in range(len(self.grupos)):
             w = self.grupos[i]
             w = self.ltrim(w)
             self.grupos[i] = self.rtrim(w)
```

Nota 1: Lo único para notar aquí es el uso de '\' al final del renglón en `removerExcepciones`. Esto lo utilizamos únicamente con fines estéticos, ya que nos permite dividir una línea en vez de tener una única línea muy larga. Python ignora el salto de línea cuando encuentra el caracter '\'.

Agregando una interfaz gráfica

Para crear una interfaz gráfica usaremos Tkinter, el cual ya ha sido introducido brevemente en la sección de Programación manejada por eventos. En este caso, la interfaz gráfica será un poco más sofisticada, ya que utilizaremos otros controles gráficos ("*widgets*") que Tkinter nos ofrece.

Rehaciendo nuestra clase Documento

Antes de llegar a este punto necesitaremos modificar nuestra clase Documento. La presente versión muestra el resultado en la pantalla como parte del método "analizar". Sin embargo, esto no nos conviene si planeamos agregar una interfaz gráfica. En su lugar, haremos que el método `analizar` guarde los resultados en los contadores, para que nosotros accedamos a ellos cuando sea necesario. Para realizar este cambio, simplemente partimos el método `mostrarResultados()` en dos partes: `generarResultados()` que calculará los valores y los guardará en los contadores, e `imprimirResultados()` que los mostrará en la pantalla.

Finalmente, deberemos modificar el método `Analizar` para que llame a a `generarResultados()`, y la parte principal para que llame a `imprimirResultados()` después de `Analizar`. Con estos cambios, el código seguirá funcionando como antes, al menos respecto del que lo use desde la línea de comando. Otros programadores deberán hacer pequeños cambios en `imprimirResultados()` después de usar `Analizar`, pero esto nos es tan grave.

El código revisado es el que sigue:

```
def generarResultados(self):
        self.c_palabras = len(self.grupos)
        oraciones, frases = 0,0
        for c in self.limites:
            oraciones = oraciones + self.c_puntuacion[c]
        self.c_oracion = oraciones
        for c in self.c_puntuacion.keys():
            frases = frases + self.c_puntuacion[c]
        self.c_frase = frases

def imprimirResultados(self):
        print self.format % (self.archivo, self.c_parrafo,
                             self.c_renglon, self.c_oracion,
                             self.c_frase, self.c_palabras)
        print "Se utilizaron los siguientes signos de puntuación:"
        for i in self.c_puntuacion.keys():
            print "\t%s\t:\t%4d" % (i,self.c_puntuacion[i])
```

and:

```
if __name__ == "__main__":
     if len(sys.argv) <> 2:
        print "Uso: python documento.py "
        sys.exit()
     else:
        try:
           D = DocumentoHTML(sys.argv[1])
           D.Analizar()
```

```
            D.imprimirResultados()
        except:
            print "Error al analizar el archivo: %s" % sys.argv[1]
```

Ahora ya estamos preparados para agregar nuestra interfaz gráfica.

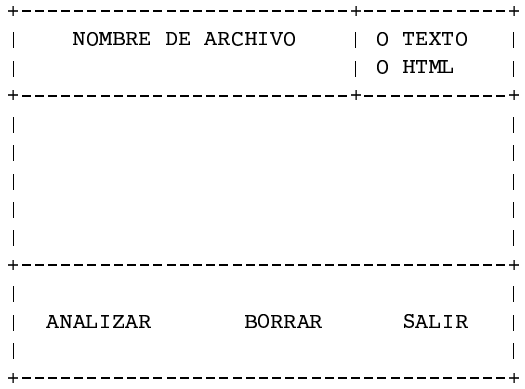
Diseñando una Interfaz Gráfica

El primer paso es tratar de visualizar cómo va a verse la interfaz. Si debemos especificar el nombre de un archivo, necesitaremos un control *Edit* o *Entry*. Como también debemos especificar si se trata de un documento de texto o HTML, utilizaremos un control *Radiobutton* (botón de radio). Estos controles deben estar agrupados para que se note que están relacionados.

El próximo requisito es buscar un display para nuestros resultados. Podríamos optar por una serie de controles *Label* (etiqueta) donde se mostrara el estado de cada uno de los contadores. En su lugar, prefiero utilizar un simple control *Text* donde colocar cadenas de caracteres. Esta última elección lo acerca más al modo de línea de comando, pero esto es más una cuestión de gustos y depende del que la diseñe.

Finalmente necesitamos un medio para iniciar el análisis y para cerrar la aplicación. Dado que utilizamos un control "Text", también sería util disponer de un medio para borrar el display. Todas estas opciones pueden representarse mediante el control *Button* (botón).

Este sería el esquema de nuestra interfaz gráfica:



Ahora escribiremos el código en pequeños pasos:

```
from Tkinter import *
import documento

################### DEFINICIONES DE CLASES ######################
class Gramatica(Frame):
    def __init__(self):
        Frame.__init__(self)
        self.type = 2 # creamos variable con valor por defecto
        self.crearInterfaz()
```

Aquí hemos importado los módulos "documento" y "Tkinter". Para el segundo hemos dispuesto que todos sus nombres estén visibles dentro del módulo actual, mientras que en el caso del primero deberemos prefijar los nombres con la referencia "documento.".

Hemos definido también un método `__init__` que llama a su vez al método `Frame.__init__` de la superclase, para asegurarnos de que Tkinter esté correctamente instalado internamente. Luego creamos un atributo que guardará el tipo de documento con el cual trabajaremos. Finalmente llamamos al método `crearInterfaz` que crea todos los controles gráficos que necesitamos.

```
    def crearInterfaz(self):
         # Primero creamos la ventana de la aplicación y su título
        fApp = Frame(0)
        fApp.wm_title = "Análisis gramatical"


        # Ahora la información del archivo: nombre y tipo
        fFile = Frame(fApp)
        Label(fFile, text="Nombre de archivo: ").pack(side="left")
        self.eName = Entry(fFile)
        self.eName.insert(INSERT,"test.htm")
        self.eName.pack(side="left", padx=5)


        # para mantener alineados los botones de radio con el nombre
        # necesitamos otro marco
        fType = Frame(fFile, borderwidth=1, relief=SUNKEN)
        self.rText = Radiobutton(fType, text="TEXTO",
                                 variable = self.type, value=2,
```

```
                            command=self.hacerTexto)
        self.rText.pack(side=TOP)
        self.rHTML = Radiobutton(fType, text="HTML",
                            variable=self.type, value=1,
                            command=self.hacerHTML)
        self.rHTML.pack(side=TOP)
        # TEXTO es la selección por defecto
        self.rText.select()
        fType.pack(side="right", padx=3)
        fFile.pack(side="top", fill=X)


        # el cuadro de texto mostrará los resultados, le aplicamos "pad" para colocarle un borde
        self.txtBox = Text(fApp, width=60, height=10)
        self.txtBox.pack(side=TOP, padx=3, pady=3)

        # finalmente agregamos un par de botones para realizar el trabajo
        fButts = Frame(fApp)
        self.bAnal = Button(fButts,
                            text="Analizar",
                            command=self.analizarEvento)
        self.bAnal.pack(side=LEFT, anchor=W, padx=50, pady=2)
        self.bReset = Button(fButts,
                             text="Resetear",
                             command=self.resetear)
        self.bReset.pack(side=LEFT, padx=10)
        self.bQuit = Button(fButts,
                            text="Salir",
                            command=self.eventoSalir)
        self.bQuit.pack(side=RIGHT, anchor=E, padx=50, pady=2)

        fButts.pack(side=BOTTOM, fill=X)
        fApp.pack()
```

No voy a explicar todo lo que hemos hecho; en su lugar te recomiendo que le des una ojeada al tutorial de Tkinter en el sitio de Python. Es una excelente introducción a Tkinter, y además es muy completo. El principio general es que uno crea controles gráficos ("widgets") a partir de una serie de clases y proveyéndole una serie de parámetros, luego el control gráfico es "empacado" (*pack*) en un marco (*frame*) que lo contiene.

El otro punto importante a notar es el uso de marcos subsidiarios para ubicar correctamente los botones de radio y los botones comunes. Los botones de radio (*Radiobuttons*) también aceptan las opciones `variable` y `value`; la primera unifica los botones de radio al especificar una misma variable externa (`self.type`), mientras que la segunda establece un valor único para cada botón de radio. También es importante notar el uso de la opción `command=xxx` que se pasa a los botones de control: estos son los métodos que serán llamados por Tkinter cada vez que se presione el botón. El código de estas llamadas es el siguiente:

```
    ################## CONTROL DE EVENTOS ####################
    # llegó la hora de morir...
    def eventoSalir(self):
        import sys
        sys.exit()


    # volvemos a los valores por defecto
    def resetear(self):
        self.txtBox.delete(1.0, END)
        self.rText.select()


    # colocar los valores de los botones de radio
    def hacerTexto(self):
        self.type = 2

    def hacerHTML(self):
        self.type = 1
```

Estos métodos son bastante sencillos y supongo que fácilmente comprensibles. El último controlador de eventos es el que realiza el análisis:

```
    # Crear un documento apropiado y analizarlo,
    # luego mostramos los resultados en el formulario
```

```
    def analizarEvento(self):
        archivo = self.eName.get()
        if archivo == "":
                self.txtBox.insert(END,"\nFalta el nombre de archivo!\n")
                return
        if self.type == 2:
                doc = documento.DocumentoTexto(archivo)
        else:
                doc = documento.DocumentoHTML(archivo)
        self.txtBox.insert(END, "\nAnalizando...\n")
        doc.Analizar()
        str = doc.format % (archivo,
                            doc.c_parrafo, doc.c_renglon,
                            doc.c_oracion, doc.c_frase, doc.c_palabras)
        self.txtBox.insert(END, str)
```

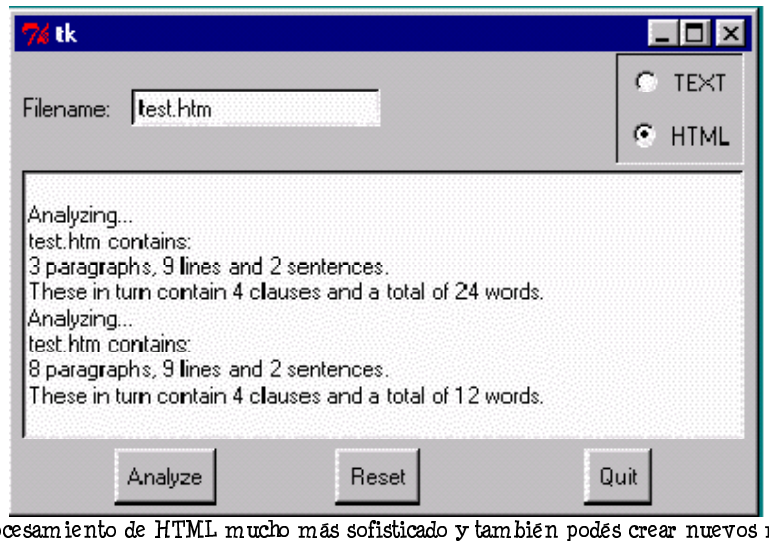
Nuevamente te darás cuenta de qué es lo que ocurre aquí. Los puntos fundamentales son los siguientes:

- Revisa si hay un nombre de archivo válido antes de crear el objeto documento.
- Utiliza el valor de `self.type` provisto por los botones de radio para determinar el tipo de Documento a crear.
- Agrega al final (argumento `END` del método `insert`) los resultados obtenidos. Esto implica que podemos analizar varias veces y comparar los resultados nuevos con los anteriores (esta es una ventaja respecto del uso de etiquetas).

Lo único que nos falta ahora es crear una instancia del objeto Aplicación y lanzar el bucle de ejecución:

```
myApp = Gramatica()
myApp.mainloop()
```

Veamos los resultados de nuestro programa en MS Windows mostrando el análisis de un archivo primero en modo Texto y luego en HTML:



Esto es todo. Podés hacer el procesamiento de HTML mucho más sofisticado y también podés crear nuevos módulos para distintos tipos de documentos. También podés cambiar la apariencia utilizando etiquetas en vez de que aparezcan todos los resultados en un cuadro de texto. Por nuestra parte y según los objetivos que nos hemos trazado, estamos listos. La siguiente sección ofrece algunas ideas para seguir practicando según tus propias aspiraciones. Lo importante es que lo disfrutes, ah, y recordá siempre: la computadora es tonta!

Anterior Referencias Contenidos

Si tenés sugerencias o dudas podés enviar un email en inglés a: alan.gauld@btinternet.com o en español a: manilio@xoommail.com