Un buen estilo de codificación

¿De qué nos ocuparemos?
Nuevos usos de los comentarios, cómo diagramar el código utilizando el sangrado para mejorar la legibilidad del programa, y una introducción al uso de módulos para guardar los programas.

Comentarios

Ya hemos hablado acerca de los comentarios en el segundo capítulo dedicado a las secuencias. Sin embargo, hay otros asuntos importantes relacionados con los comentarios, que pasaremos a considerar ahora:

Información sobre la historia de las versiones

Siempre es recomendable crear un encabezado al comienzo de cada archivo, en el cual se incluyan detalles tales como la fecha de creación, el autor, la versión del programa y una descripción general del contenido. Este bloque debe aparecer como un comentario:

```
##############################
# Módulo:   Spam.py
# Autor:    A.J.Gauld
# Fecha:    1999/09/03
# Versión:  Borrador 0.4
#
# Descripción: Este módulo provee un objeto Spam que puede ser combinado
# con cualquier otro objeto Comida para producir una variedad interesante
# de deliciosos platos.
#
##############################
# Log:
# 1999/09/01    AJG - Se crea el archivo
# 1999/09/02    AJG - Se arregla el error en los precios de la comida
# 1999/09/02    AJG - Ahora funciona bien
# 1999/09/03    AJG - Se agrega método para hervir (cf Change Req #1234)
##############################
import sys, string, comida
...
```

Deshabilitar código con comentarios

Esta técnica se usa generalmente para diferenciar una sección del programa que no funciona correctamente. Por ejemplo, supongamos que un programa lee ciertos datos de un archivo, los procesa y guarda los resultados en el mismo archivo. Si los resultados no son los esperados, sería útil poder evitar que se ejecute esta sección hasta poder descubrir la falla. Una solución radical sería directamente borrar la sección, pero también podemos convertir las líneas en comentarios, preservándolas sin mayor efecto en el programa dado que los comentarios no se ejecutan:

```
datos = readData(archivo)
for item in datos:
    resultado.append(calcularResultado(item))
printResultado(resultado)
######################
# Comentamos la sección hasta que solucionemos el problema con calcularResultado
# for item in resultado:
#     dataFile.save(item)
######################
print 'El programa terminó.'
```

Una vez que hayamos solucionado el error simplemente borramos los marcadores de comentario y el código vuelve a estar activo nuevamente.

Cadenas de Documentación

Todos los lenguajes permiten utilizar comentarios para documentar qué realiza una función o un módulo, pero unos pocos como Python y Smalltalk van un paso más allá y permiten documentar las funciones de un modo tal que el mismo lenguaje puede proveer una ayuda interactiva en la fase de programación. En Python esto se realiza mediante una cadena de documentación de la forma: """documentación""":

```
class Spam:
    """Un tipo de carne que puede combinarse con otras comidas

    Puede utilizarse con otros alimentos para preparar comidas muy interesantes.
    Viene con muchísimos nutrientes y puede cocinarse utilizando
    diferentes técnicas"""
```

```
    def __init__(self):
        ...

print Spam.__doc__
```

Nota: Podemos acceder a la *cadena de documentación* imprimiendo la variable especial `__doc__` . Los módulos, las funciones y las clases/métodos aceptan cadenas de documentación. Podés probar lo siguiente:

```
import sys
print sys.__doc__
```

Sangrado

Esto es fuente de acalorados debates en los círculos de programación, y parecería que cada programador/a tiene su propia idea acerca de cuál es la mejor manera de tabular el código. Se han realizado estudios que demuestran que al menos hay ciertos factores que realmente son importantes para una mayor legibilidad del programa y exceden un criterio simplemente estético.

La razón del debate es sencilla: en la mayor parte de los lenguajes el uso de las tabulaciones es un criterio puramente estético y se utilizan como ayuda para el lector. Sin embargo, en otros lenguajes como Python, el uso de tabulaciones es esencial para que el programa funcione correctamente. Veamos algunos ejemplos que aclararán lo anterior:

```
FOR I = 1 TO 10
    PRINT I
NEXT I
```

Es exactamente igual a:

```
FOR I = 1 TO 10
PRINT I
NEXT I
```

al menos en el intérprete de BASIC. El primer ejemplo es más sencillo de leer ya que el sangrado aclara mejor las estructuras.

El punto importante en esto es que el sangrado debe reflejar la estructura lógica del código de modo tal que visualmente siga el flujo del programa. De este modo, es de gran ayuda que el sangrado se visualice como un bloque:

```
XXXXXXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
    XXXXXXXXXXXXXXX
```

que es ciertamente más claro que:

```
XXXXXXXXXXXXXXXXXXXX
  XXXXX
    XXXXXXXXXXXX
    XXXXXXXXXXXX
    XXXXXXXXXXXX
  XXXXX
```

dado que claramente indica de que se trata de un sólo bloque. Estudios recientes han demostrado que la comprensión aumenta cuando el sangrado refleja la estructura lógica del bloque. En ejemplos breves y sencillos como los que hemos visto hasta ahora, este tema podría parecer de relativa importancia. Sin embargo, cuando empieces a escribir programas con cientos o miles de líneas, te darás cuenta de la importancia de que dichas estructuras sean bien legibles.

Los nombres de las variables

Los nombres de las variables que hemos utilizado hasta ahora no han tenido demasiado sentido ya que tenían un propósito didáctico. En general es recomendable que el nombre de la variable refleje claramente el contenido que almacena. Por ejemplo, en el ejercicio sobre las tablas de multiplicación utilizamos la variable "multiplicador" para indicar qué tabla estábamos mostrando. Esto tiene mucho más sentido que haber utilizado el nombre "m", que también hubiera funcionado correctamente y es más fácil de escribir.

Esto implica una lucha eterna entre lo comprensible y lo sencillo. En general la mejor opción es utilizar nombres breves pero sensatos. Un nombre muy largo se vuelve confuso y es difícil utilizarlo con consistencia a lo largo del programa (podríamos haber utilizado la variable `la_tabla_que_estamos_mostrando` en lugar de `multiplicador`, pero es muy largo y realmente no es mucho más claro que el nombre elegido).

Programación Modular

Nos ocuparemos en detalle de este tema más adelante, pero ahora veremos los conceptos fundamentales y, sobre todo, el uso de módulos para guardar nuestro trabajo (hasta ahora todos los programas se han perdido una vez que salimos de Python).

Aunque el intérprete interactivo de Python (>>>) es realmente muy útil para probar ideas de una forma rápida, tan pronto lo cerramos habremos perdido todo lo que tipeamos en él. Pronto desearemos ser capaces de escribir un programa más complejo y ejecutarlo una y

otra vez. Para realizar esto en Python creamos un archivo de texto con la extensión `.py` (esto es una convención, en realidad se puede usar cualquier extensión que uno quiera, pero creo que en este punto es una buena idea mantener dicha convención). Luego podemos ejecutar el programa desde la línea de comando tipeando simplemente:

```
$ python spam.py
```

Donde "spam.py" es el nombre del programa escrito en Python.

Nota para usuarios de Unix: La primera línea de un *listado* en un archivo debe contener la secuencia `#!` seguida por la ruta completa de Python en tu sistema (esto lo podés averiguar tipeando `$ which python` en el shell).

En mi sistema la línea es:

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

Esto permitirá ejecutar el archivo sin tener que cargar Python al mismo tiempo: `$ spam.py`
Esta línea no produce ningún efecto ni ningún daño en Windows/Mac, por lo que estos usuarios pueden ponerla en sus programas para que puedan ejecutarse también en sistemas Unix.

Nota para usuarios de Windows: Con el Explorador de Windows se puede crear una asociación entre los archivos terminados en `.py` y el intérprete de Python. Esto permite ejecutar los programas haciendo un doble click sobre el ícono del archivo. Esto se realiza automáticamente con la instalación de Python. Podés comprobar esto buscando algún archivo terminado en .py e intentando ejecutarlo. Si el programa se inicia (incluso con un mensaje de error) la asociación está creada.

La otra ventaja de usar archivos para guardar programas es que uno puede editar y corregir los errores sin tener que retipear todo el fragmento, o en IDLE, subir con el cursor hasta la línea deseada, reseleccionar el código y corregirlo. IDLE puede cargar también archivos y ejecutarlos desde el menú 'Edit|Run module'.

De ahora en más no aparecerá más el prompt `>>>` en los ejemplos. Doy por sentado que escribís los programas en un archivo separado y luego los ejecutás con IDLE o desde la línea de comando (mi opción favorita).

Para recordar

- Los comentarios son útiles también para evitar que se ejecute temporariamente un fragmento de código, lo que nos sirve para probar o corregir un programa.
- Los comentarios pueden proveer un encabezado indicativo con la historia de las distintas versiones del programa.
- Las cadenas de documentación pueden ser utilizadas para brindar información acerca de los módulos y los objetos en tiempo de ejecución.
- El sangrado de los bloques de código incrementa la legibilidad y hace patente la estructura lógica del programa.
- Podemos guardar un programa y ejecutarlo más tarde si lo escribimos en un archivo en lugar de en el prompt '>>>'. Luego puede ejecutarse directamente mediante `$ python progname.py` en la línea de comando o haciendo doble click sobre el ícono del archivo en el Explorador de Windows.

Anterior Próxima Contenido

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