Programación Modular

¿Qué es un módulo?

El cuarto elemento fundamental de nuestro viaje por la programación está representado por la *programación modular*. En realidad, esta técnica no es estrictamente necesaria ya que con lo que hemos visto hasta ahora podés crear programas interesantes y útiles. Sin embargo, cuando los programas se vuelven más complejos y más extensos, se hace más difícil rastrear los errores y analizar su funcionalidad. Con este fin necesitamos disponer de una técnica que nos permita concentrarnos realmente en los problemas que debemos resolver con nuestro programa y abstraernos de los detalles particulares y de la parafernalia técnica que hace funcionar a la computadora. De algún modo esta tarea la cumplen Python, BASIC, etc. ya que con sus funciones incorporadas y predeterminadas evitan que nosotros debamos tratar directamente con el hardware de la computadora para realizar diversas tareas típicas como leer un archivo, comprobar que una tecla se ha presionado, etc.

El objetivo de la programación modular es *extender* las capacidades predeterminadas de un lenguaje mediante porciones de código empacadas en diferentes módulos, los cuales pueden ser fácilmente insertados en nuestros programas. La primera forma de programación modular fue la *subrutina* que era un bloque de código al cual se podía saltar (similar a la instrucción `GOTO` que hemos mencionado antes) y una vez ejecutado el bloque, el programa devolvía el control a la instrucción siguiente a la que había producido la llamada. Esta técnica modular se conoce con el nombre de *procedimiento* o *función*. En Python y en otros lenguajes, el término *módulo* tiene un significado especial que explicaremos más adelante; mientras tanto veamos más en detalle las funciones.

La utilización de funciones

Antes de analizar cómo crear una función veamos cómo se utilizan las diversas funciones predeterminadas que vienen incorporadas en cualquier lenguaje de programación (conocidas en general como *bibliotecas*).

Ya hemos utilizado varias funciones y hemos listado otras en la sección dedicada a los operadores. Ahora veremos qué tienen en común todas ellas y cómo podemos utilizarlas en nuestros programas.

La estructura básica de una función es la siguiente:

```
Valor = algunaFunción(unArgumento, otroArgumento, etc...)
```

Esto significa que la variable Valor obtiene su valor al llamar a la función. Una función puede aceptar varios, uno o ningún *argumento*. La función utiliza estos argumentos como variables internas e incluso puede llamar a otras funciones. Veamos algunos ejemplos utilizando nuestros lenguajes:

BASIC: MID$(cad$,n,m)

Esta función devuelve `m` caracteres de la cadena `cad$` contados a partir del `n`. (Recordemos que el signo '$' implica una cadena en BASIC).

```
tiempo$ = "TARDES NOCHES DÍAS"
PRINT "Buenas";MID$(tiempo$,7,6)
```

Este programa imprimirá entonces "Buenas NOCHES".

BASIC: ENVIRON$(cad$)

Esta función devuelve la *Variable del Entorno* especificada en `cad$`.

```
PRINT ENVIRON$("PATH")
```

Imprime la ruta actual definida en el DOS (usualmente en el archivo autoexec.bat).

Tcl: llength L

Devuelve el tamaño de la lista `L`.

```
set a {"primero" "segundo" "tercero"} # una lista de tres elementos
puts [llength $a]  # devuelve el valor '3'
```

Nota: En Tcl prácticamente todo es una función (o un comando como prefiere llamarlo Tcl). Esto produce una sintaxis un tanto extraña para el usuario pero muy sencilla para la computadora. Recordemos que Tcl es la abreviatura de *Tool Control Language (Lenguaje de Control de Herramientas)* y fue diseñado para servir como un *lenguaje de Macros* a la manera del *Visual Basic for Applications* (VBA) utilizado en los productos de Microsoft. Python puede funcionar de esta misma manera, pero la diferencia es que Tcl fue creado principalmente con ese objetivo.

Python: pow(a,n)

```
a = 2   #  usamos 2 como número base
for n in range(0,11):
   print pow(a,n)    # elevamos 2 a la potencia n, es decir entre 0 y 10
```

Aquí primero generamos los valores de `n` entre `0` y `10` y llamamos a la función predeterminada `pow()` a la cual le pasamos dos argumentos: `a` y `n`. En cada llamada los nuevos valores de `a` y `n` son sustituidos y un nuevo resultado se imprime en la pantalla.

Nota: El operador exponencial `**` es equivalente a la función `pow()`.

Python: dir(m)

Otra función muy útil predeterminada en Python es `dir`, la cual devuelve una lista de nombres válidos (en general de funciones) dentro del módulo `m`. Veamos un ejemplo con funciones predeterminadas:

```
print dir(__builtin__)
```

Antes de avanzar más con este tema, veamos con mayor detalle el funcionamiento de los módulos en Python.

La utilización de módulos

Python es un lenguaje extremadamente extensible (igual que Tcl) en tanto uno puede agregarle nuevas 'habilidades' al `importar` los módulos. En seguida veremos cómo se crean dichos módulos, pero primero probemos algunos de los módulos estándar que se incluyen en la distribución de Python.

sys

Ya hemos utilizado el módulo `sys` para cerrar Python. Este módulo presenta otras muchas funciones de gran utilidad. Para acceder a estas debemos importar el módulo mediante la instrucción `import sys`:

```
import sys # ponemos las funciones a disposición del programa
sys.exit() # prefijamos con 'sys'
```

Si sabemos de antemano que usaremos con gran frecuencia estas funciones en nuestro programa y que dichas funciones no llevan el mismo nombre que otras que ya hayamos importado o creado, entonces podemos utilizar otra variante:

```
from sys import *  # importamos todos los nombres en sys
exit() # ahora no es necesario usar el prefijo 'sys'
```

Los otros módulos y sus contenidos

Podemos importar y utilizar cualquiera de los módulos de la misma manera que lo hicimos con sys, tanto los originales de Python como otros que puedas haber creado vos mismo o bajado de Internet. En un momento veremos cómo hacer esto. Aquí incluyo una breve lista de los módulos más importantes que ofrece Python:

Nombre del módulo	Descripción
`sys`	Interactúa con el sistema de Python: • exit() - salida! • argv - acceso a los argumentos de la línea de comando • path - acceso a la ruta del sistema • ps1 - cambia el prompt '>>>' de Python
`os`	Interactúa con el sistema operativo: • open - abre un archivo • system - ejecuta un comando del sistema • mkdir - crea un directorio • getcwd - busca el *directorio de trabajo actual*
`string`	Manipulación de cadenas • atoi/f/l - convierte una cadena a integer/float/long • find - busca una subcadena • split - separa en 'palabras' • upper/lower - convierte a mayúsculas/minúsculas
`re`	Manipulación de cadenas como expresiones regulares de Unix • search - busca un patrón en cualquier lugar de la cadena • match - busca sólo al comienzo de la cadena • split - separa una cadena en campos según un delimitador • sub,subn - sustitución de cadenas
`math`	Acceso a diversas funciones matemáticas: • sin,cos etc - funciones trigonométricas • log,log10 - logaritmos naturales y decimales • ceil,floor - redondeo hacia arriba y hacia abajo • pi, e - constantes naturales
`time`	Funciones de hora y fecha • time - hora actual (expresada en segundos) • gmtime - convertir hora en segundos a UTC (GMT) • localtime - convertir a hora local • mktime - inverso de localtime • sleep - detener el programa por *n* segundos

Esto es sólo la punta del iceberg. Hay docenas de módulos incluidos en la distribución de Python, y muchos otros que podés conseguir en Internet. Revisá la documentación para averiguar acerca de la programación en Internet, gráficos, bases de datos, etc.

Lo más importante de todo esto es darse cuenta que la mayor parte de los lenguajes de programación incluyen estas funciones básicas, ya como predeterminadas, ya como parte de una biblioteca. Siempre hay que revisar la documentación antes de crear una nueva función, ya que es posible que la función exista de antemano. Y esto nos lleva a...

Definiendo nuestras propias funciones

Ahora ya sabemos cómo usar las funciones existentes, pero ¿cómo creamos nuestras propias funciones? Simplemente basta con *declararlas*. Esto significa que debemos escribir una instrucción que le indique al intérprete que hemos creado un bloque de código que ha de actuar como una función y que debe estar disponible para ser llamado por el programa.

Para ver cómo funciona crearemos una función que imprima la tabla de multiplicación que nosotros solicitemos por medio de un argumento. En BASIC:

```
SUB MULTI (N%)
FOR I = 1 TO 12
    PRINT I; "x"; N%; "="; I * N%
NEXT I
END SUB
```

Y podemos *llamar* a la función de esta manera:

```
PRINT "Esta es la tabla del siete:"
MULTI(7)
```

Nota: Definimos un *parámetro* llamado `N%` y lo pasamos como *argumento* con el valor de 7. La variable local `N%` dentro de la función tomó el valor 7 cuando la llamamos. Podemos definir la cantidad de parámetros que querramos en la definición de la función, pero cuando se llame a esta función debemos proveer los valores para cada uno de estos parámetros. Algunos lenguajes de programación permiten definir *valores por defecto* para un parámetro, de tal modo que si no se incluye un valor determinado en la llamada, la función toma el valor por defecto. Veremos esta funcionalidad en Python.

In Python la función MULTI sería así:

```
def multi(n):
    for i in range(1,13):
        print "%d x %d = %d" % (i, n, i*n)
```

Y la llamamos de esta manera:

```
print "Esta es la tabla del 9"
multi(9)
```

Fijate que estas funciones no devuelven ningún valor (son en realidad lo que algunos lenguajes denominan *procedimientos*); de hecho, BASIC utiliza la palabra `SUB` en lugar de `FUNCTION`. Esta es una abreviatura de *subrutina*, un término un tanto añejo de la programación en Assembler que significa en BASIC una función que no devuelve un valor. En contraste, Python utiliza el término `def` que es la abreviatura de 'define' (definir) y da por sentado que lo que sigue es una función.

¿Te acordás que mencioné el uso de valores por defecto? Un uso sensato de estos valores podría verse en una función que devolviera el día de la semana correspondiente a una determinada fecha. Si llamamos a la función sin argumentos la fecha corresponde al día de hoy, de lo contrario deberemos proveer una fecha particular. El programa sería así:

```
# si día es -1 => hoy
def diasSemana(DiaNum = -1):
    dias = ['Lunes','Martes',
            'Miércoles','Jueves',
            'Viernes', 'Sábado', 'Domingo']

    # comprobamos valor por defecto
    if DiaNum == -1:
        # Usamos las funciones del módulo TIME para obtener la fecha actual
        # revisá la tabla más arriba y la documentación oficial del módulo
        import time
        laFecha = time.localtime(time.time())
        DiaNum = laFecha[6]
    return dias[DiaNum]
```

Nota: Sólo tendremos que usar el módulo time si el valor del parámetro es por defecto, por lo cual diferimos la operación de importarlo hasta que lo necesitamos. Esta técnica produce una ligera mejoría en la performance del programa si no necesitamos utilizar el módulo (con el consiguiente retardo en la carga del mismo dentro de nuestro código).

Entonces podemos llamar a esta función así:

```
print "Hoy es: %s" % diasSemana()
# recordá que en el lenguaje de las computadoras empezamos a contar desde 0
# y en este caso tomamos al lunes como el primer día de la semana.
print "El tercer día es: %s" % diasSemana(2)
```

Volvamos a nuestro viejo ejemplo de la multiplicación...

Nuevo desafío: ahora deseamos definir una función que nos devuelva los valores de la tabla de multiplicación en un vector de números. En BASIC lo haríamos así:

```
FUNCTION MULTI% (N%)
    DIM VALORES(12) AS INTEGER
    FOR I = 1 to 12
        VALORES(I) = I*N%
    NEXT I
    RETURN VALORES
END FUNCTION
```

Y en Python:

```
def multi(n):
    # creamos una lista vacía nueva
    valores = []
    for i in range(1,13):
        valores.append(i*n)
    return valores
```

Como te darás cuenta esto no tiene una gran utilidad, ya que es más sencillo calcular simplemente `i*n` según se requiera. Pero con suerte habras comprendido la razón de esto. Una función más práctica podría ser una que devolviera la cantidad de palabras en una cadena. Podríamos utilizarla para contar las palabras en un archivo, sumando los totales de cada renglón. El código de una función tal sería:

```
def numpalabras(s):
    list = split(s) # cada elemento de la lista es una palabra
    return len(list) # devuelve el número de elementos en la lista

for renglon in archivo:
    total = total + numpalabras(renglon) # acumulamos el total de cada renglón
print "El archivo tiene %d palabras" % total
```

Si ejecutaste este código te habrás dado cuenta de que no funciona. Lo que he presentado aquí es una técnica de diseño muy común que consiste en bosquejar el código sin preocuparnos demasiado en si lo estamos utilizando correctamente. En general esto se conoce como *Pseudo Código* o de una manera más formal *Program Description Language (PDL), "Lenguaje de Descripción de Programas"*.

Cuando más adelante hayamos visto el capítulo sobre archivos y el manejo de cadenas, volveremos sobre este ejemplo para codificarlo correctamente.

Las funciones en Tcl

En este punto Tcl difiere bastante del resto de los lenguajes. La mayor parte de los lenguajes de programación incluyen un conjunto de palabras claves como `for, while, if/else` etc., mientras que Tcl presenta estas palabras claves como comandos o funciones. Esto produce el efecto -interesante, confuso y muy poderoso- de permitirnos redefinir las estructuras de control predeterminadas de este modo:

```
set i 3
while {$i < 10} {
    puts $i
    set i [expr $i + 1]
    }
```

Como es de esperar este fragmento imprime los números del tres al nueve (1 menos que 10). Pero definamos ahora nuestra propia versión del comando while:

```
proc while {x y} {
  puts "Este es mi while"
}

set i 3
while {$i < 10} {
    puts $i
    set i [expr $i + 1]
    }
```

Esto no hace más que imprimir un mensaje. La expresión y la secuencia de comandos son ignorados porque Tcl trata a todos ellos como parámetros de la función while, y la función while, por su parte, los espera pero los ignora. Así ves cómo se definen los procedimientos

en Tcl y como podemos abusar de esta característica para crear programas muy confusos. Por favor, no lo hagas a menos que tengas una buena razón para ello!

Creando nuestros propios módulos

Ahora llegó el momento de crear nuestras propias funciones y llamarlas desde otras partes del programa. Esto es importante no sólo porque nos permite ahorrar mucho tipeo sino también porque vuelve a nuestros programas mucho más sencillos de seguir, ya que podemos olvidarnos de los detalles después de crear la función que los agrupa. (Este principio de "empaque" de las partes complejas de un programa en una serie de funciones se denomina *ocultamiento de la información* por razones obvias.) Ahora, ¿cómo usamos estas funciones en otros programas? Muy sencillo: creamos un módulo.

Los módulos en Python

Un módulo en Python no es nada especial. Es un simple archivo de texto lleno de instrucciones en Python. En general, estas instrucciones son definiciones de funciones. Para habilitar el módulo escribimos:

```
from sys import *
```

de esta manera efectivamente copiamos los contenidos del módulo `sys.py` en nuestro programa, como si fuera una operación de "cortar y pegar" (no es exactamente así, pero el concepto es claro). En algunos lenguajes de programación como C++, el intérprete literalmente copia los archivos de módulo en el programa actual cuando así lo es requerido.

Recapitulemos un poco: creamos un módulo al escribir un archivo de Python que contiene las funciones que deseamos reutilizar en otros programas. Luego simplemente importamos nuestro módulo exactamente igual que lo hacemos con los módulos estándar que se incluyen en la distribución de Python. ¿Sencillo, no? Bueno, hagámoslo entonces.

Copiá la función que se encuentra a continuación y guardála en un archivo con el nombre de tablamul.py

```
def imprime_tabla(multiplicador):
    print "--- Esta es la tabla del %d ---" % multiplicador
    for n in range(1,13):
        print "%d x %d = %d" % (n, multiplicador, n*multiplicador)
```

En la línea de comando de Python escribí:

```
>>> import tablamul
>>> tablamul.imprime_tabla(12)
```

Aquí llegamos: hemos creado un módulo y lo hemos utilizado.

Nota importante: Si no ejecutaste a Python desde el mismo directorio en el cual guardaste el archivo tablamul.py, es probable que Python no haya podido encontrar el archivo y haya generado un mensaje de error. Para solucionar esto podés crear una variable de entorno llamada PYTHONPATH que contiene una lista de directorios válidos donde buscar los módulos (tanto los estándar como los nuevos que hayas creado). *La creación de variables de entorno es una operación específica de cada plataforma, que doy por sentado que sabrás hacer o al menos podrás averiguar por vos mismo.*

Los módulos en BASIC y Tcl

¿Qué pasa con el BASIC? Esto es más complicado... En QBASIC y en las variedades más antiguas no existe el concepto de módulo. En estos casos es necesario cortar y pegar manualmente en nuestro programa el código de programas anteriores por medio de un editor de textos. Sin embargo, en Visual Basic el concepto de módulo sí existe, y uno puede cargar un módulo en el *Integrated Development Environment (IDE) (Entorno Integrado de Desarrollo)* a través del menú `File|Open Module...`. Hay algunas restricciones al tipo de acciones que uno puede realizar en un módulo de BASIC, pero no profundizaré en este tema, ya que no nos ocuparemos del Visual Basic. (Nota: si querés experimentar un poco con esta variedad de Basic, hay una versión reducida de prueba de Visual Basic que podés bajar gratis del sitio de Microsoft: www.microsoft.com).

Finalmente, Tcl, como siempre(!), toma un camino un tanto ecléctico, pero no menos interesante, respecto de la reutilización de módulos (o *bibliotecas*, como prefiere llamarlas).

En el nivel más simple uno puede crear un archivo de funciones en Tcl igual que lo hicimos con Python, y luego poner en nuestro programa al archivo de funciones como `source` (fuente). Esta intrucción ordena al intérprete leer el archivo y ponerlo a disposición del programa principal para su utilización. Pero existe una opción más interesante:

Podemos crear nuestros archivos como vimos más arriba, ponerlos en un directorio y luego ejecutar el comando `mk_index`. Este crea un índice con todas las funciones y los archivos del directorio especificado. Luego en nuestro programa llamamos a las funciones y el intérprete de Tcl al no encontrar la función en el programa, la buscará automáticamente en el índice de funciones que fue creado previamente. Luego simplemente importa el código relevante de la biblioteca y ejecuta la función. Esta característica de Tcl se denomina *autoloading (carga automática)*.

Una vez que se coloca la función como fuente (source), permanece siempre disponible, por lo cual es muy poca la pérdida de performance del programa. El único problema es que debemos evitar crear dos funciones con el mismo nombre.

En nuestro siguiente paso nos dedicaremos a los archivos y al tratamiento de cadenas, y luego volveremos a nuestro programa prometido que cuenta palabras en un archivo. De hecho, crearemos un módulo de funciones útiles para el tratamiento de texto.

Anterior Próxima Contenido

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