Recursividad

Nota: Este es un tema bastante avanzado, y para la mayor parte de las aplicaciones no será necesario que conozcas absolutamente nada de él. En algunas ocasiones es tan útil que se vuelve invaluable su utilización, por esta razón lo incluyo en este tutorial para que lo conozcas. Por favor, no enloquezcas si no lo entendés de entrada.

¿De qué se trata?

A pesar de lo que dije antes de que los bucles son uno de los pilares fundamentales de la programación, sin embargo, es posible construir programas sin utilizarlos. Algunos lenguajes como Lisp, no tienen una construcción explícita de bucles tal como `FOR`, `WHILE`, etc., sino que utilizan una técnica de programación conocida como *recursividad*. Esta resulta ser una técnica muy poderosa para la solución de determinados problemas, así que ahora le daremos un vistazo.

La recursividad simplemente significa aplicar una función como parte de la definición de esa misma función.

La clave de funcionamiento es que **obligatoriamente debe existir una condición terminal** con el objeto de que la función se bifurque hacia una resolución no recursiva en algún punto. De lo contrario, la función entra en un bucle infinito y nunca finaliza.

Veamos un ejemplo sencillo. La función matemática factorial se define como el producto de todos los números hasta el argumento inclusive. El factorial de 1 es 1. Si pensamos un poco, nos daremos cuenta de que tenemos otra manera de expresar esta función: el factorial de N es igual a N veces el factorial de N-1.

Por lo tanto:

```
1! = 1
2! = 1 x 2 = 2
3! = 1 x 2 x 3 = 2! x 3 = 6
N! = 1 x 2 x 3 x .... (N-2) x (N-1) x N = (N-1)! x N
```

En Python podemos expresar esto así:

```
def factorial(n):
    if n == 1:
        return 1
    else:
        return n * factorial(n-1)
```

Dado que restamos uno de N cada vez y comprobamos si es igual a uno, la función no es infinita y puede completarse correctamente.

Escribir la función del factorial sin utilizar la recursividad implica el desarrollo de mucho más código. Necesitamos crear una lista de todos los números desde el 1 al N y luego iterar en la lista multiplicando el total actual por el item siguiente. Podés probar hacerlo como un ejercicio y comparálo con la función definida más arriba.

Recursividad en las listas

El otro campo donde la recursividad es muy útil es en el procesamiento de listas. Siempre que podamos evaluar si una lista está vacía y generar una lista menos su primer elemento, podremos usar fácilmente la recursividad en ellas.

Pensá en el caso trivial de tener que imprimir cada elemento de una lista de cadenas usando una función "imprimeLista":

```
def imprimeLista(L):
    if L:
        print L[0]
        # acerca de [1:] releé en el Manual de Referencia de Python acerca de la técnica de "slicing"
        imprimeLista(L[1:])
```

Si L es verdadero -es decir, no está vacía- imprimimos el primer elemento y luego procesamos el resto de la lista.

Para una lista simple esta solución es bastante trivial, ya que podríamos resolverla con un bucle común. Pero si la lista fuera muy compleja y contuviera otras listas dentro de sí misma, la situación sería muy diferente. Si podemos evaluar si un item es una lista, fácilmente podremos llamar a imprimeLista() de manera recursiva. De lo contrario, simplemente la imprimimos. Veamos un poco cómo funciona esto:

```
def imprimeLista(L):
    # si está vacía no hacemos nada
    if not L: return
    # si es una lista llamamos a imprimeLista para el primer elemento
    if type(L[0]) == type([]):
        imprimeLista(L[0])
    else: #no es una lista, simplemente imprimimos
        print L[0]
    # ahora procesamos el resto de L
    printList(L[1:])
```

Si intentás hacer esto utilizando un bucle convencional, te darás cuenta que no es sencillo. La recursividad permite realizar tareas muy complejas de una manera bastante más simple.

Sin embargo, hay un pequeño problema (¡como siempre!): la recursividad suele demandar mucha memoria cuando se utilizan grandes estructuras de datos. Por esta razón, a veces es más seguro utilizar los procedimientos convencionales, sobre todo si se dispone de poca memoria o las estructuras de datos son muy grandes.

Bueno, ahora haremos otro salto hacia lo desconocido al introducirnos en la Programación Orientada a Objetos.

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