Recursión en Python

Recordemos el problema planteado anteriormente:

> *Supongamos que vamos a la casa de un amigo pero se nos olvidó traer la dirección completa y sólo tenemos el nombre de la calle en la que vive. Al llegar nos damos cuenta que es un pasaje con 8 casas a cada lado ¿Qué podemos hacer para no perder el viaje? Asumiendo que los vecinos no se conocen entre sí, y que no tenemos forma de contactarlo previamente, lo mejor sería simplemente ir y golpear en la primera casa, y preguntar si nuestro amigo vive ahí. En caso de que la respuesta sea no, tendríamos que* ***repetir*** *el proceso con las siguientes casas, hasta encontrar la de nuestro amigo.*

En la clase anterior aprendimos la forma en que podíamos resolver este problema, planteando un **ciclo `while`**. Sin embargo, esta no es la única posibilidad. En el problema anterior, supongamos que a la casa del amigo vamos cuatro personas. Para encontrar más velozmente a nuestro amigo, ¿no sería mejor, en vez de ir todos preguntando casa por casa, qué cada uno de nosotros preguntara en dos casas distintas?

De esta forma cada uno de nosotros está **resolviendo instancias más pequeñas** del **problema de búsqueda**. Luego podemos **reunirnos nuevamente** y conocer cuál es la casa de nuestro amigo.

Funciones recursivas

Para ver como esto se puede llevar a Python miremos el siguiente programa.

Recursivo.py

```
1. def recursivo(n):
2.     if n == 0 :
3.         return 1
4.     return n * recursivo(n - 1)
5.
6.
7. #
8. # Bloque principal
9. #
10.
11.print recursivo(5)
```

¿Qué tiene de especial? Si nos fijamos, la función `recursivo()` **se llama a sí misma**, pero cambiando el valor del parámetro actual. A esta forma de **repetir** las sentencias del cuerpo de una función se le conoce como **recursión**.

Recuerda

Como primera aproximación, podemos decir que una función recursiva es aquella que se define en **términos de sí misma**: "*para entender la recursividad hay que entender la recursividad*". Para entender mejor la definición, miremos la siguiente imagen de una lata de polvos de hornear ROYAL:

Si miramos la etiqueta es autorreferente, pues su etiqueta es la misma lata, y dentro de esa lata, está nuevamente una lata y así sucesivamente.

Pregunta 1

Trabaja ahora con tu grupo en responder la pregunta 1 de la actividad.

Si nos fijamos en el recuadro, tanto en la foto como en la definición de recursividad escrita en cursiva, existe el problema de que hipotéticamente podemos realizar el proceso de repetir la definición de recursividad, o encontrar la etiqueta dentro de la lata de la etiqueta **infinitamente**, es decir, no hay forma de detenerla. Sin embargo, el programa `Recursivo.py`, si se detiene. ¿Qué hace la función `recursivo(n)` para no llamarse a sí misma infinitamente?

La respuesta a esta pregunta es sencilla. Si miramos cuidadosamente veremos que, en cada llamada, **mientras** el valor de `n` sea positivo (no cumple la condición del `if` de la línea 2), el valor de `n` disminuye en `1` (llamada de la línea 4). Por otra parte, la función **deja de llamarse a sí misma** cuando `n` llega al valor `0` (sentencia de retorno de la línea 3 si la condición del `if` de la línea 2 se cumple).

Importante

Una función recursiva necesariamente debe contener por lo menos estos dos elementos:

Regla de recursión:

```
n * recursivo (n - 1)
```

Condición de borde:

```
if(n == 0):
    return 1
```

Ejemplo 1

Se define el **factorial de un entero no negativo** *n*, denotado por *n*!, como la multiplicación de todos los números naturales menores o iguales que *n.*

Además se define que:

$$0! = 1$$

Podemos ver fácilmente que el caso de 0! es la condición de borde de la función factorial de un entero no negativo *n*.

Veamos ahora la regla de recursión. Sabemos que:

$$n! = n * (n-1) * (n-2) * \ldots * 2 * 1$$

Que es lo mismo que decir:

$$\begin{aligned} n! &= n * (n-1)! = \\ &= n * (n-1) * (n-2)! = \\ &= n * (n-1) * \ldots * 2 * 1 * 0! \end{aligned}$$

Así tenemos que:

$$n! = \begin{cases} 1, & n = 0 \\ n * (n-1)!, & n > 0 \end{cases}$$

Que es la operación que realiza el programa `Recursivo.py`.

Existen muchas **funciones recursivas** en las matemáticas y en la naturaleza, y son la base de la **inducción matemática**. Probablemente ya has conocido muchos ejemplos en álgebra a través de las demostraciones por medio de inducción o en cálculo a través de **ecuaciones de recurrencia**.

Preguntas 2

Junto a tu grupo responde la pregunta 2 de la actividad.

Es importante tener en cuenta la forma en que se **evalúan** las funciones recursivas. Como es necesario conocer el **caso base** para poder conocer el resultado, el computador no puede entregar el resultado final **hasta** llegar hasta el caso base, por lo que se hace necesario almacenar temporalmente en memoria los valores ya conocidos en espera que ocurra el caso base.

Para la función anterior, si hacemos la llamada `recursivo(3)`, en Python se resolverá internamente de la siguiente manera:

- Se guarda el 3 en memoria y se efectúa la llamada `recursivo(2)`.
- Se guarda el 2 en memoria y se efectúa la llamada `recursivo(1)`.
- Se guarda el 1 en memoria y se efectúa la llamada `recursivo(0)`.
- Como se encuentra la condición de borde, se devuelve un `1`.
- Se multiplica el 1 guardado por el resultado anterior `(1 * 1 = 1)`.
- Se multiplica el 2 guardado por el resultado anterior `(2 * 1 = 3)`.
- Se multiplica el 3 guardado por el resultado anterior `(3 * 2 = 6)`.
- Se entrega el resultado.

Pregunta 3

Es hora de desarrollar la pregunta 3 de la actividad de hoy.

Debemos notar que cada llamada recursiva genera una **nueva instancia** de ejecución de la función, es decir, se utiliza **un espacio en memoria distinto**, con **variables distintas** y resultados **intermedios distintos**, aún si para el usuario la función tiene el mismo

nombre. Por ejemplo para el caso de `Recursivo.py`, con llamada `recursivo(5)`, las instancias de ejecución en memoria podrían lucir como se muestra en la figura 1.

Figura 1: Ejecución en memoria de `recursivo(5)`

Podemos ver que las instancias de ejecución de la función `recursivo()` están almacenadas **simultáneamente** en memoria, con exactamente los mismos nombres de función y de variables, pero **valores distintos**, donde cada instancia, excepto la última, está a la espera que se resuelva `recursivo(n - 1)`, para poder entregar su propio resultado.

Notemos también que la comunicación entre las instancias de ejecución de una función es **limitada al retorno de un valor cuando concluyen**, y por lo tanto **no podemos acceder** a valores de variables o parámetros en otras ejecuciones pendientes.

Más importante, como todas las instancias de ejecución pendientes se mantienen en memoria, las soluciones recursivas suelen ser **computacionalmente muy costosas** en términos de **memoria**, y a veces también en términos de **velocidad**, y no es extraño que se produzcan **errores** por **agotamiento** de los recursos del computador. Python pone un **límite a la cantidad de recursiones** que una función puede hacer, intentando evitar que se pueda sobrepasar la capacidad de la memoria disponible.

Importante

Por esta razón, y a **diferencia** de los ciclos `while` que pueden tener cuerpos tan complejos cómo se quiera, las funciones recursivas tienden a ser muy simples, sin almacenar grandes cantidades de datos, por lo que también resultan muy elegantes.

Con esta salvedad, debemos saber que **cualquier repetición** de ejecución de sentencias es posible de ser implementada con una **iteración** (ciclo `while`) o con una **función recursiva**.

Por ejemplo, al igual que con los ciclos `while`, una función recursiva nos sirve para **insistir** en una interacción con el usuario cuando éste ingresa valores inválidos, como para el caso de los menús. El programa `EjemploMenuRecursivo.py` muestra una versión recursiva que obliga al usuario a elegir las opciones 1, 2 ó 3.

EjemploMenuRecursivo.py

```
#
# Función que maneja el menú.
# Entrada:
# Salida: opción ingresada por el usuario (valor entero);
#         se asegura que la opción elegida es válida
#
def eligeOpcionMenu():
    print
    print "MENU"
    print "===="
    print
    print "Opción 1: Saludar"
    print "Opción 2: Despedir"
    print "Opción 3: Salir del programa"
    print

    opcion = input("Elija una opción: ")

    if opcion == 1 or opcion == 2 or opcion == 3:
        return int(opcion)

    return eligeOpcionMenu()
```

Notemos que la **condición de borde** en la versión recursiva de `eligeOpcionMenu()` es que el usuario **ingrese una opción válida**. Cuando esto ocurre, la función simplemente devuelve esta opción como resultado. Pero **mientras esto no ocurra**, la función devuelve **el resultado de llamar a la función nuevamente** para pedir que se escoja una opción del menú.

Otro ejemplo de la equivalencia funcional entre iteración y recursión es el programa FactorialIterativo.py, que implementa una versión iterativa de la función factorial, que entrega los mismos resultados que la versión recursiva en el programa `Recursivo.py`.

Iterativo.py

```
def iterativo(n):
    contador = 1
    resultado = 1

    while contador <= n:
        resultado = resultado * contador
        contador= contador + 1

    return resultado
```

Podemos ver que `iterativo(n)` es más difícil de **modelar**, es decir, es más difícil que se nos "ocurra" hacerlo de este modo, pero es menos costoso, en términos de **memoria y procesamiento**, que la versión recursiva. ¿Por qué?

Pregunta 4

Con tu grupo trabaja en el desarrollo de la pregunta 4.