6.7. Diseño de programas con funciones

Hemos aprendido a diseñar funciones, cierto, pero puede que no tengas claro qué ventajas nos reporta trabajar con ellas. El programa de integración numérica que hemos desarrollado en la sección anterior podría haberse escrito directamente así:

integral_8.py — integral.py

```
a = float(raw_input('Inicio del intervalo: '))
b = float(raw_input('Final del intervalo: '))
n = int(raw_input('Número de rectángulos: '))

if n == 0:
    sumatorio = 0
else:
    deltax = (b-a) / float(n)
    sumatorio = 0
    for i in range(n):
        sumatorio += deltax * (a + i * deltax) ** 2

print 'La integral de x**2 entre %f y %f es (aprox) %f' % (a, b, sumatorio)
```

Este programa ocupa menos líneas y hace lo mismo, ¿no? Sí, así es. Con programas pequeños como éste apenas podemos apreciar las ventajas de trabajar con funciones. Imagina que el programa fuese mucho más largo y que hiciese falta aproximar el valor de la integral definida de x^2 en tres o cuatro lugares diferentes; entonces sí que sería una gran ventaja haber definido una función: habiendo escrito el procedimiento de cálculo una vez podríamos ejecutarlo cuantas veces quisiéramos mediante simples invocaciones. No sólo eso, habríamos ganado en legibilidad.

6.7.1. Ahorro de tecleo

Por ejemplo, supón que en un programa deseamos leer tres números enteros y asegurarnos de que sean positivos. Podemos proceder repitiendo el bucle correspondiente tres veces:

Evita las llamadas repetidas

En nuestra última versión del programa `biseccion.py` hay una fuente de ineficiencia: $f(c)$, para un c fijo, se calcula 3 veces por iteración.

`biseccion.py`

```
def f(x):
    return x**2 - 2*x -2

def biseccion(a, b, epsilon):
    c = (a + b) / 2.0
    while f(c) != 0 and b - a > epsilon:
        if f(a)*f(c) > 0:
            a = c
        elif f(b)*f(c) > 0:
            b = c
        c = (a + b) / 2.0
    return c

print 'El cero está en:', biseccion(0.5, 3.5, 1e-5)
```

Llamar a una función es costoso: Python debe dedicar un tiempo a gestionar la pila de llamadas apilando una nueva trama activación de función (y ocupar, en consecuencia, algo de memoria), copiar las referencias a los valores de los argumentos en los parámetros formales, efectuar nuevamente un cálculo que ya hemos hecho y devolver el valor resultante.

Una optimización evidente del programa consiste en no llamar a $f(c)$ más que una vez y almacenar el resultado en una variable temporal que usaremos cada vez que deberíamos haber llamado a $f(c)$:

biseccion_4.py

`biseccion.py`

```
def f(x):
    return x**2 - 2*x -2

def biseccion(a, b, epsilon):
    c = (a + b) / 2.0
    fc = f(c)
    while fc != 0 and b - a > epsilon:
        if f(a)*fc > 0:
            a = c
        elif f(b)*fc > 0:
            b = c
        c = (a + b) / 2.0
        fc = f(c)
    return c

print 'El cero está en:', biseccion(0.5, 3.5, 1e-5)
```

`lee_positivos.py`

```
a = int(raw_input('Dame un número positivo: '))
while a < 0:
    print 'Has cometido un error: el número debe ser positivo'
    a = int(raw_input('Dame un número positivo: '))

b = int(raw_input('Dame otro número positivo: '))
while b < 0:
```

```
8     print 'Has␣cometido␣un␣error:␣el␣número␣debe␣ser␣positivo'
9     b = int(raw_input('Dame␣otro␣número␣positivo:␣'))
10
11  c = int(raw_input('Dame␣otro␣número␣positivo:␣'))
12  while c < 0:
13    print 'Has␣cometido␣un␣error:␣el␣número␣debe␣ser␣positivo'
14    c = int(raw_input('Dame␣otro␣número␣positivo:␣'))
```

O podemos llamar tres veces a una función que lea un número y se asegure de que sea positivo:

lee_positivos.py

```
1  def lee_entero_positivo(texto):
2    numero = int(raw_input(texto))
3    while numero < 0:
4      print 'Ha␣cometido␣un␣error:␣el␣número␣debe␣ser␣positivo'
5      numero = int(raw_input(texto))
6    return numero
7
8  a = lee_entero_positivo('Dame␣un␣número␣positivo:␣')
9  b = lee_entero_positivo('Dame␣otro␣número␣positivo:␣')
10 c = lee_entero_positivo('Dame␣otro␣número␣positivo:␣')
```

Hemos reducido el número de líneas, así que hemos tecleado menos. Ahorrar tecleo tiene un efecto secundario beneficioso: reduce la posibilidad de cometer errores. Si hubiésemos escrito mal el procedimiento de lectura del valor entero positivo, bastaría con corregir la función correspondiente. Si en lugar de definir esa función hubiésemos replicado el código, nos tocaría corregir el mismo error en varios puntos del programa. Es fácil que, por descuido, olvidásemos corregir el error en uno de esos lugares y, sin embargo, pensásemos que el problema está solucionado.

6.7.2. Mejora de la legibilidad

No sólo nos ahorramos teclear: un programa que utiliza funciones es, por regla general, más legible que uno que inserta los procedimientos de cálculo directamente donde se utilizan; bueno, eso siempre que escojas nombres de función que describan bien qué hacen éstas. Fíjate en que el último programa es más fácil de leer que el anterior, pues estas tres líneas son autoexplicativas:

```
8   a = lee_entero_positivo('Dame␣un␣número␣positivo:␣')
9   b = lee_entero_positivo('Dame␣otro␣número␣positivo:␣')
10  c = lee_entero_positivo('Dame␣otro␣número␣positivo:␣')
```

6.7.3. Algunos consejos para decidir qué debería definirse como función: análisis descendente y ascendente

Las funciones son un elemento fundamental de los programas. Ahora ya sabes *cómo* construir funciones, pero quizá no sepas *cuándo* conviene construirlas. Lo cierto es que no podemos decírtelo: no es una ciencia exacta, sino una habilidad que irás adquiriendo con la práctica. De todos modos, sí podemos darte algunos consejos.

1. Por una parte, *todos los fragmentos de programa que vayas a utilizar en más de una ocasión son buenos candidatos a definirse como funciones*, pues de ese modo evitarás tener que copiarlos en varios lugares. Evitar esas copias no sólo resulta más cómodo: también reduce considerablemente la probabilidad de que cometas errores, pues acabas escribiendo menos texto. Además, si cometes errores y has de

corregirlos o si has de modificar el programa para ampliar su funcionalidad, siempre será mejor que el mismo texto no aparezca en varios lugares, sino una sola vez en una función.

2. *Si un fragmento de programa lleva a cabo una acción que puedes nombrar o describir con una sola frase, probablemente convenga convertirlo en una función*. No olvides que los programas, además de funcionar correctamente, deben ser legibles. Lo ideal es que el programa conste de una serie de definiciones de función y un programa principal breve que las use y resulte muy legible.

3. *No conviene que las funciones que definas sean muy largas*. En general, una función debería ocupar menos de 30 o 40 líneas (aunque siempre hay excepciones). *Una función no sólo debería ser breve, además debería hacer una única cosa... y hacerla bien*. Deberías ser capaz de describir con *una sola frase* lo que hace cada una de tus funciones. Si una función hace tantas cosas que explicarlas todas cuesta mucho, probablemente harías bien en dividir tu función en funciones más pequeñas y simples. Recuerda que puedes llamar a una función desde otra.

El proceso de identificar acciones complejas y dividirlas en acciones más sencillas se conoce como *estrategia de diseño descendente* (en inglés, «top-down»). La forma de proceder es ésta:

- analiza primero qué debe hacer tu programa y haz un esquema que explicite las diferentes acciones que debe efectuar, pero sin entrar en el detalle de cómo debe efectuarse cada una de ellas;
- define una posible función por cada una de esas acciones;
- analiza entonces cada una de esas acciones y mira si aún son demasiado complejas; si es así, aplica el mismo método hasta que obtengas funciones más pequeñas y simples.

Ten siempre presente la relación de datos que necesitas (serán los parámetros de la función) para llevar a cabo cada acción y el valor o valores que devuelve.

Una estrategia de diseño alternativa recibe el calificativo de *ascendente* (en inglés, «bottom-up») y consiste en lo contrario:

- detecta algunas de las acciones más simples que necesitarás en tu programa y escribe pequeñas funciones que las implementen;
- combina estas acciones en otras más complejas y crea nuevas funciones para ellas;
- sigue hasta llegar a una o unas pocas funciones que resuelven el problema.

Ahora que empiezas a programar resulta difícil que seas capaz de anticiparte y detectes a simple vista qué pequeñas funciones te irán haciendo falta y cómo combinarlas apropiadamente. Será más efectivo que empieces siguiendo la metodología descendente: ve dividiendo cada problema en subproblemas más y más sencillos que, al final, se combinarán para dar solución al problema original. Cuando tengas mucha más experiencia, probablemente descubrirás que al programar sigues una estrategia híbrida, ascendente y descendente a la vez. Todo llega. Paciencia.