4.2.5. El bucle for-in

Hay otro tipo de bucle en Python: el bucle **for-in**, que se puede leer como «para todo elemento de una serie, hacer...». Un bucle **for-in** presenta el siguiente aspecto:

```
for variable in serie de valores:
    acción
    acción
    ...
    acción
```

Veamos cómo funciona con un sencillo ejemplo:

saludos.py

```
1 for nombre in ['Pepe', 'Ana', 'Juan']:
2     print 'Hola, %s.' % nombre
```

Fíjate en que la relación de nombres va encerrada entre corchetes y que cada nombre se separa del siguiente con una coma. Se trata de una *lista* de nombres. Más adelante estudiaremos con detalle las listas. Ejecutemos ahora el programa. Por pantalla aparecerá el siguiente texto:

```
Hola, Pepe.
Hola, Ana.
Hola, Juan.
```

Se ha ejecutado la sentencia más indentada una vez por cada valor de la serie de nombres y, con cada iteración, la variable *nombre* ha tomado el valor de uno de ellos (ordenadamente, de izquierda a derecha).

En el capítulo anterior estudiamos el siguiente programa:

potencias_2.py — potencias.py

```
numero = int(raw_input('Dame un número: '))

print '%d elevado a %d es %d' % (numero, 2, numero ** 2)
print '%d elevado a %d es %d' % (numero, 3, numero ** 3)
print '%d elevado a %d es %d' % (numero, 4, numero ** 4)
print '%d elevado a %d es %d' % (numero, 5, numero ** 5)
```

Ahora podemos ofrecer una versión más simple:

potencias.py — potencias.py

```
numero = int(raw_input('Dame un número: '))

for potencia in [2, 3, 4, 5]:
    print '%d elevado a %d es %d' % (numero, potencia, numero ** potencia)
```

El bucle se lee de forma natural como «para toda *potencia* en la serie de valores 2, 3, 4 y 5, haz...».

EJERCICIOS

▶ 117 Haz un programa que muestre la tabla de multiplicar de un número introducido por teclado por el usuario. Aquí tienes un ejemplo de cómo se debe comportar el programa:

```
Dame un número: 5
5 x 1 = 5
5 x 2 = 10
5 x 3 = 15
5 x 4 = 20
5 x 5 = 25
5 x 6 = 30
5 x 7 = 35
5 x 8 = 40
5 x 9 = 45
5 x 10 = 50
```

▶ 118 Realiza un programa que proporcione el desglose en billetes y monedas de una cantidad entera de euros. Recuerda que hay billetes de 500, 200, 100, 50, 20, 10 y 5 € y monedas de 2 y 1 €. Debes «recorrer» los valores de billete y moneda disponibles con uno o más bucles **for-in**.

▶ 119 Haz un programa que muestre la raíz *n*-ésima de un número leído por teclado, para *n* tomando valores entre 2 y 100.

El último ejercicio propuesto es todo un desafío a nuestra paciencia: teclear 99 números separados por comas supone un esfuerzo bárbaro y conduce a un programa poco elegante.

Es hora de aprender una nueva función predefinida de Python que nos ayudará a evitar ese tipo de problemas: la función *range* (que en inglés significa «rango»). En principio, *range* se usa con dos argumentos: un *valor inicial* y un *valor final* (con matices).

```
>>> range(2, 10) ↲
[2, 3, 4, 5, 6, 7, 8, 9]
>>> range(0, 3) ↲
[0, 1, 2]
>>> range(-3, 3) ↲
[-3, -2, -1, 0, 1, 2]
```

Observa que la lista devuelta contiene todos los enteros comprendidos entre los argumentos de la función, incluyendo al primero *pero no al último*.

La función *range* devuelve una lista de números enteros. Estudia este ejemplo:

contador_con_for.py

```
1 for i in range(1, 6):
2     print i
```

Al ejecutar el programa, veremos lo siguiente por pantalla:

```
1
2
3
4
5
```

La lista que devuelve *range* es usada por el bucle **for-in** como serie de valores a recorrer.

El último ejercicio propuesto era pesadísimo: ¡nos obligaba a escribir una serie de 99 números! Con *range* resulta muchísimo más sencillo. He aquí la solución:

raices_2.py

```
1 numero = float(raw_input('Dame un número: '))
2
3 for n in range(2, 101):
4     print 'la raíz %d-ésima de %f es %f' % (numero, n, numero**(1.0/n))
```

(Fíjate en que *range* tiene por segundo argumento el valor 101 y no 100: recuerda que el último valor de la lista es el segundo argumento *menos uno*.)

Podemos utilizar la función *range* con uno, dos o tres argumentos. Si usamos *range* con un argumento estaremos especificando únicamente el último valor (más uno) de la serie, pues el primero vale 0 por defecto:

```
>>> range(5) ↲
[0, 1, 2, 3, 4]
```

Si usamos tres argumentos, el tercero permite especificar un *incremento* para la serie de valores. Observa en estos ejemplos qué listas de enteros devuelve *range*:

```
>>> range(2, 10, 2) ↲
[2, 4, 6, 8]
>>> range(2, 10, 3) ↲
[2, 5, 8]
```

Fíjate en que si pones un incremento negativo (un *decremento*), la lista va de los valores altos a los bajos. Recuerda que con *range* el último elemento de la lista no llega a ser el valor final

```
>>> range(10, 5, -1) ↲
[10, 9, 8, 7, 6]
>>> range(3, -1, -1) ↲
[3, 2, 1, 0]
```

Así pues, si el tercer argumento es negativo, la lista finaliza en el valor final *más uno* (y no menos uno).

Finalmente, observa que es equivalente utilizar *range* con dos argumentos a utilizarla con un valor del incremento igual a 1.

```
>>> range(2, 5, 1) ↲
[2, 3, 4]
>>> range(2, 5) ↲
[2, 3, 4]
```

EJERCICIOS

▶ 120 Haz un programa que muestre, en líneas independientes, todos los números pares comprendidos entre 0 y 200 (ambos inclusive).

▶ 121 Haz un programa que muestre, en líneas independientes y en orden inverso, todos los números pares comprendidos entre 0 y 200 (ambos inclusive).

▶ 122 Escribe un programa que muestre los números pares positivos entre 2 y un número cualquiera que introduzca el usuario por teclado.

Obi Wan

Puede resultar sorprendente que *range*(*a*, *b*) incluya todos los números enteros comprendidos entre *a* y *b*, pero sin incluir *b*. En realidad la forma «natural» o más frecuente de usar *range* es con un sólo parámetro: *range*(*n*) que devuelve una lista con los *n* primeros números enteros incluyendo al cero (hay razones para que esto sea lo conveniente, ya llegaremos). Como incluye al cero y hay *n* números, no puede incluir al propio número *n*. Al extenderse el uso de *range* a dos argumentos, se ha mantenido la «compatibilidad» eliminando el último elemento. Una primera ventaja es que resulta fácil calcular cuántas iteraciones realizará un bucle *range*(*a*, *b*): exactamente *b* − *a*. (Si el valor *b* estuviera incluido, el número de elementos sería *b* − *a* + 1.)

Hay que ir con cuidado, pues es fácil equivocarse «por uno». De hecho, equivocarse «por uno» es tan frecuente al programar (y no sólo con *range*) que hay una expresión para este tipo de error: un error Obi Wan (Kenobi), que es más o menos como suena en inglés «off by one» (pasarse o quedarse corto por uno).