Más herramientas para control de flujo

Además de la sentencia **`while`** que acabamos de introducir, Python soporta las sentencias de control de flujo que podemos encontrar en otros lenguajes, con algunos cambios.

La sentencia

Tal vez el tipo más conocido de sentencia sea el **`if`**. Por ejemplo:

```
>>> x = int(input("Ingresa un entero, por favor: "))
Ingresa un entero, por favor: 42
>>> if x < 0:
...      x = 0
...      print('Negativo cambiado a cero')
... elif x == 0:
...      print('Cero')
... elif x == 1:
...      print('Simple')
... else:
...      print('Más')
...
'Mas'
```

Puede haber cero o más bloques **`elif`**, y el bloque **`else`** es opcional. La palabra reservada '**`elif`**' es una abreviación de 'else if', y es útil para evitar un sangrado excesivo. Una secuencia **`if`** ... **`elif`** ... **`elif`** ... sustituye las sentencias `switch` o `case` encontradas en otros lenguajes.

La sentencia

La sentencia **`for`** en Python difiere un poco de lo que uno puede estar acostumbrado en lenguajes como C o Pascal. En lugar de siempre iterar sobre una progresión aritmética de números (como en Pascal) o darle al usuario la posibilidad de definir tanto el paso de la iteración como la condición de fin (como en C), la sentencia **`for`** de Python itera sobre los ítems de cualquier secuencia (una lista o una cadena de texto), en el orden que aparecen en la secuencia. Por ejemplo:

```
>>> # Midiendo cadenas de texto
... palabras = ['gato', 'ventana', 'defenestrado']
>>> for p in palabras:
...     print(p, len(p))
...
gato 4
ventana 7
defenestrado 12
```

Si necesitás modificar la secuencia sobre la que estás iterando mientras estás adentro del ciclo (por ejemplo para borrar algunos ítems), se recomienda que hagas primero una copia. Iterar sobre una secuencia no hace implícitamente una copia. La notación de rebanada es especialmente conveniente para esto:

```
>>> for p in palabras[:]:  # hace una copia por rebanada de toda la lista
...    if len(p) > 6:
...        palabras.insert(0, p)
...
```

```
>>> palabras
['defenestrado', 'ventana', 'gato', 'ventana', 'defenestrado']
```

La función

Si se necesita iterar sobre una secuencia de números, es apropiado utilizar la función integrada **`range()`**, la cual genera progresiones aritméticas:

```
>>> for i in range(5):
...     print(i)
...
0
1
2
3
4
```

El valor final dado nunca es parte de la secuencia; `range(10)` genera 10 valores, los índices correspondientes para los ítems de una secuencia de longitud 10. Es posible hacer que el rango empiece con otro número, o especificar un incremento diferente (incluso negativo; algunas veces se lo llama 'paso'):

```
range(5, 10)
   5 through 9

range(0, 10, 3)
   0, 3, 6, 9

range(-10, -100, -30)
  -10, -40, -70
```

Para iterar sobre los índices de una secuencia, podés combinar **`range()`** y **`len()`** así:

```
>>> a = ['Mary', 'tenia', 'un', 'corderito']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 Mary
1 tenia
2 un
3 corderito
```

En la mayoría de los casos, sin embargo, conviene usar la función **`enumerate()`**, mirá *Técnicas de iteración*.

Algo extraño sucede si mostrás un `range`:

```
>>> print(range(10))
range(0, 10)
```

De muchas maneras el objeto devuelto por **`range()`** se comporta como si fuera una lista, pero no lo es. Es un objeto que devuelve los ítems sucesivos de la secuencia deseada cuando iterás sobre él, pero realmente no construye la lista, ahorrando entonces espacio.

Decimos que tal objeto es *iterable*; esto es, que se lo puede usar en funciones y construcciones que esperan algo de lo cual obtener ítems sucesivos hasta que se termine. Hemos visto que la declaración **`for`** es un *iterador* en ese sentido. La función **`list()`** es otra; crea listas a partir de iterables:

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

Más tarde veremos más funciones que devuelven iterables y que toman iterables como entrada.

Las sentencias , , y en lazos

La sentencia **`break`**, como en C, termina el lazo **`for`** o **`while`** más anidado.

Las sentencias de lazo pueden tener una cláusula `else` que es ejecutada cuando el lazo termina, luego de agotar la lista (con **`for`**) o cuando la condición se hace falsa (con **`while`**), pero no cuando el lazo es terminado con la sentencia **`break`**. Se ejemplifica en el siguiente lazo, que busca números primos:

```
>>> for n in range(2, 10):
...     for x in range(2, n):
...         if n % x == 0:
...             print(n, 'es igual a', x, '*', n/x)
...             break
...     else:
...         # sigue el bucle sin encontrar un factor
...         print(n, 'es un numero primo')
...
2 es un numero primo
3 es un numero primo
4 es igual a 2 * 2
5 es un numero primo
6 es igual a 2 * 3
7 es un numero primo
8 es igual a 2 * 4
9 es igual a 3 * 3
```

(Sí, este es el código correcto. Fijate bien: el `else` pertenece al ciclo **`for`**, **no** al **`if`**.)

Cuando se usa con un ciclo, el `else` tiene más en común con el `else` de una declaración **`try`** que con el de un **`if`**: el `else` de un **`try`** se ejecuta cuando no se genera ninguna excepción, y el `else` de un ciclo se ejecuta cuando no hay ningún `break`. Para más sobre la declaración **`try`** y excepciones, mirá *Manejando excepciones*.

La declaración **`continue`**, también tomada de C, continua con la siguiente iteración del ciclo:

```
>>> for num in range(2, 10):
...     if num % 2 == 0:
...         print("Encontré un número par", num)
...         continue
...     print("Encontré un número", num)
Encontré un número par 2
Encontré un número 3
Encontré un número par 4
Encontré un número 5
Encontré un número par 6
Encontré un número 7
Encontré un número par 8
Encontré un número 9
```

La sentencia

La sentencia **`pass`** no hace nada. Se puede usar cuando una sentencia es requerida por la sintáxis pero el programa no requiere ninguna acción. Por ejemplo:

```
>>> while True:
...     pass  # Espera ocupada hasta una interrupción de teclado (Ctrl+C)
...
```