Estructuras de datos

Este capítulo describe algunas cosas que ya aprendiste en más detalle, y agrega algunas cosas nuevas también.

Más sobre listas

El tipo de dato lista tiene algunos métodos más. Aquí están todos los métodos de los objetos lista:

`list.append(x)`
Agrega un ítem al final de la lista. Equivale a `a[len(a):] = [x]`.

`list.extend(iterable)`
Extiende la lista agregándole todos los ítems del iterable. Equivale a `a[len(a):] = iterable`.

`list.insert(i, x)`
Inserta un ítem en una posición dada. El primer argumento es el índice del ítem delante del cual se insertará, por lo tanto `a.insert(0, x)` inserta al principio de la lista, y `a.insert(len(a), x)` equivale a `a.append(x)`.

`list.remove(x)`
Quita el primer ítem de la lista cuyo valor sea *x*. Es un error si no existe tal ítem.

`list.pop([i])`
Quita el ítem en la posición dada de la lista, y lo devuelve. Si no se especifica un índice, `a.pop()` quita y devuelve el último ítem de la lista. (Los corchetes que encierran a *i* en la firma del método denotan que el parámetro es opcional, no que deberías escribir corchetes en esa posición. Verás esta notación con frecuencia en la Referencia de la Biblioteca de Python.)

`list.clear()`
Quita todos los elementos de la lista. Equivalente a `del a[:]`.

`list.index(x[, start[, end]])`
Devuelve un índice basado en cero en la lista del primer ítem cuyo valor sea *x*. Levanta una excepción `ValueError` si no existe tal ítem.
Los argumentos opcionales *start* y *end* son interpetados como la notación de rebanadas y se usan para limitar la búsqueda a una subsecuencia particular de la lista. El index retornado se calcula de manera relativa al inicio de la secuencia completa en lugar de con respecto al argumento *start*.

`list.count(x)`
Devuelve el número de veces que *x* aparece en la lista.

`list.sort(key=None, reverse=False)`
Ordena los ítems de la lista in situ (los argumentos pueden ser usados para personalizar el orden de la lista, ve `sorted()` para su explicación).

`list.reverse()`
Invierte los elementos de la lista in situ.

`list.copy()`
Devuelve una copia superficial de la lista. Equivalente a `a[:]`.

Un ejemplo que usa la mayoría de los métodos de lista:

```
>>> frutas = ['naranja', 'manzana', 'pera', 'banana', 'kiwi', 'manzana', 'banana']
>>> frutas.count('manzana')
2
>>> frutas.count('mandarina')
0
>>> frutas.index('banana')
3
>>> frutas.index('banana', 4)  # Find next banana starting a position 4
6
>>> frutas.reverse()
>>> frutas
['banana', 'manzana', 'kiwi', 'banana', 'pera', 'manzana', 'naranja']
>>> frutas.append('uva')
>>> frutas
    ['banana', 'manzana', 'kiwi', 'banana', 'pera', 'manzana', 'naranja', 'uva']
>>> frutas.sort()
>>> frutas
['manzana', 'manzana', 'banana', 'banana', 'uva', 'kiwi', 'naranja', 'pera']
>>> frutas.pop()
'pera'
```

Quizás hayas notado que métodos como `insert`, `remove` o `sort`, que solo modifican a la lista, no tienen impreso un valor de retorno -- devuelven None.[5] Esto es un principio de diseño para todas las estructuras de datos mutables en Python.

Usando listas como pilas

Los métodos de lista hacen que resulte muy fácil usar una lista como una pila, donde el último elemento añadido es el primer elemento retirado ("último en entrar, primero en salir"). Para agregar un ítem a la cima de la pila, use **`append()`**. Para retirar un ítem de la cima de la pila, use **`pop()`** sin un índice explícito. Por ejemplo:

```
>>> stack = [3, 4, 5]
>>> stack.append(6)
>>> stack.append(7)
>>> stack
[3, 4, 5, 6, 7]
>>> stack.pop()
7
>>> stack
[3, 4, 5, 6]
>>> stack.pop()
6
>>> stack.pop()
5
>>> stack
[3, 4]
```

Usando listas como colas

También es posible usar una lista como una cola, donde el primer elemento añadido es el primer elemento retirado ("primero en entrar, primero en salir"); sin embargo, las listas no son eficientes para este propósito. Agregar y sacar del final de la lista es rápido, pero insertar o sacar del comienzo de una lista es lento (porque todos los otros elementos tienen que ser desplazados por uno).

Para implementar una cola, usá **`collections.deque`** el cual fue diseñado para agregar y sacar de ambas puntas de forma rápida. Por ejemplo:

```
>>> from collections import deque
>>> queue = deque(["Eric", "John", "Michael"])
>>> queue.append("Terry")           # llega Terry
>>> queue.append("Graham")          # llega Graham
>>> queue.popleft()                 # el primero en llegar ahora se va
'Eric'
>>> queue.popleft()                 # el segundo en llegar ahora se va
'John'
>>> queue                           # el resto de la cola en órden de llegada
['Michael', 'Terry', 'Graham']
```

Comprensión de listas

Las comprensiones de listas ofrecen una manera concisa de crear listas. Sus usos comunes son para hacer nuevas listas donde cada elemento es el resultado de algunas operaciones aplicadas a cada miembro de otra secuencia o iterable, o para crear una subsecuencia de esos elementos para satisfacer una condición determinada.

Por ejemplo, asumamos que queremos crear una lista de cuadrados, como:

```
>>> cuadrados = []
>>> for x in range(10):
...     cuadrados.append(x**2)
...
>>> cuadrados
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

Nota que esto crea (o sobreescribe) una variable llamada `x` que sigue existiendo luego de que el bucle haya terminado. Podemos calcular la lista de cuadrados sin ningun efecto secundario haciendo:

```
cuadrados = list(map(lambda x: x**2, range(10)))
```

o, un equivalente:

```
cuadrados = [x ** 2 for x in range(10)]
```

que es más conciso y legible.

Una lista de comprensión consiste de corchetes rodeando una expresión seguida de la declaración **`for`** y luego cero o más declaraciones **`for`** o **`if`**. El resultado será una nueva lista que sale de evaluar la expresión en el contexto de los **`for`** o **`if`** que le siguen. Por ejemplo, esta lista de comprensión combina los elementos de dos listas si no son iguales:

```
>>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

y es equivalente a:

```
>>> combs = []
>>> for x in [1,2,3]:
...     for y in [3,1,4]:
...         if x != y:
...             combs.append((x, y))
...
>>> combs
[(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Notá como el orden de los **`for`** y **`if`** es el mismo en ambos pedacitos de código.

Si la expresión es una tupla (como el `(x, y)` en el ejemplo anterior), debe estar entre paréntesis.

```
>>> vec = [-4, -2, 0, 2, 4]
>>> # crear una nueva lista con los valores duplicados
```

```
>>> [x * 2 for x in vec]
[-8, -4, 0, 4, 8]
>>> # filtrar la lista para excluir números negativos
>>> [x for x in vec if x >= 0]
[0, 2, 4]
>>> # aplica una función a todos los elementos
>>> [abs(x) for x in vec]
[4, 2, 0, 2, 4]
>>> # llama un método a cada elemento
>>> frutafresca = ['  banana', '  mora de Logan ', 'maracuya  ']
>>> [arma.strip() for arma in frutafresca]
['banana', 'mora de Logan', 'maracuya']
>>> # crea una lista de tuplas de dos como (número, cuadrado)
>>> [(x, x ** 2) for x in range(6)]
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
>>> # la tupla debe estar entre paréntesis, sino es un error
>>> [x, x ** 2 for x in range(6)]
Traceback (most recent call last):
...
    [x, x ** 2 for x in range(6)]
               ^
SyntaxError: invalid syntax
>>> # aplanar una lista usando comprensión de listas con dos 'for'
>>> vec = [[1,2,3], [4,5,6], [7,8,9]]
>>> [num for elem in vec for num in elem]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Las comprensiones de listas pueden contener expresiones complejas y funciones anidadas:

```
>>> from math import pi
>>> [str(round(pi, i)) for i in range(1, 6)]
['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

Listas por comprensión anidadas

La expresión inicial de una comprensión de listas puede ser cualquier expresión arbitraria, incluyendo otra comprensión de listas.

Considerá el siguiente ejemplo de una matriz de 3x4 implementada como una lista de tres listas de largo 4:

```
>>> matriz = [
...     [1, 2, 3, 4],
...     [5, 6, 7, 8],
...     [9, 10, 11, 12],
... ]
```

La siguiente comprensión de lista transpondrá las filas y columnas:

```
>>> [[fila[i] for fila in matriz] for i in range(4)]
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

Como vimos en la sección anterior, la lista de comprensión anidada se evalua en el contexto del `for` que lo sigue, por lo que este ejemplo equivale a:

```
>>> transpuesta = []
>>> for i in range(4):
...     transpuesta.append([fila[i] for fila in matriz])
...
>>> transpuesta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

el cual, a la vez, es lo mismo que:

```
>>> transpuesta = []
>>> for i in range(4):
...     # las siguientes 3 lineas hacen la comprensión de listas anidada
...     fila_transpuesta = []
...     for fila in matriz:
...         fila_transpuesta.append(fila[i])
...     transpuesta.append(fila_transpuesta)
...
>>> transpuesta
[[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

En el mundo real, deberías preferir funciones predefinidas a declaraciones con flujo complejo. La función `zip()` haría un buen trabajo para este caso de uso:

```
 >>> list(zip(*matriz))
[(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

Ver *Desempaquetando una lista de argumentos* para detalles en el asterisco de esta línea.

La instrucción del

Hay una manera de quitar un ítem de una lista dado su índice en lugar de su valor: la instrucción `del`. Esta es diferente del método `pop()`, el cual devuelve un valor. La instrucción `del` también puede usarse para quitar secciones de una lista o vaciar la lista completa (lo que hacíamos antes asignando una lista vacía a la sección). Por ejemplo:

```
>>> a = [-1, 1, 66.25, 333, 333, 1234.5]
>>> del a[0]
>>> a
[1, 66.25, 333, 333, 1234.5]
>>> del a[2:4]
>>> a
[1, 66.25, 1234.5]
>>> del a[:]
>>> a
[]
```

`del` puede usarse también para eliminar variables:

```
>>> del a
```

Hacer referencia al nombre `a` de aquí en más es un error (al menos hasta que se le asigne otro valor). Veremos otros usos para `del` más adelante.