Seealso

Tipos integrados

Las cadenas de texto son ejemplos de *tipos secuencias*, y soportan las operaciones comunes para esos tipos.

Tipos integrados

Las cadenas de texto soportan una gran cantidad de métodos para transformaciones básicas y búsqueda.

Tipos integrados

Aquí se da información sobre formateo de cadenas de texto con **`str.format()`**.

Tipos integrados

Aquí se describe con más detalle las operaciones viejas para formateo usadas cuando una cadena de texto o una cadena Unicode están a la izquierda del operador %.

Listas

Python tiene varios tipos de datos *compuestos*, usados para agrupar otros valores. El más versátil es la *lista*, la cual puede ser escrita como una lista de valores separados por coma (ítems) entre corchetes. Las listas pueden contener ítems de diferentes tipos, pero usualmente los ítems son del mismo tipo:

```
>>> cuadrados = [1, 4, 9, 16, 25]
>>> cuadrados
[1, 4, 9, 16, 25]
```

Como las cadenas de caracteres (y todos los otros tipos *sequence* integrados), las listas pueden ser indexadas y rebanadas:

```
>>> cuadrados[0]  # índices retornan un ítem
1
>>> cuadrados[-1]
25
>>> cuadrados[-3:]  # rebanadas retornan una nueva lista
[9, 16, 25]
```

Todas las operaciones de rebanado devuelven una nueva lista conteniendo los elementos pedidos. Esto significa que la siguiente rebanada devuelve una copia superficial de la lista:

```
>>> cuadrados[:]
[1, 4, 9, 16, 25]
```

Las listas también soportan operaciones como concatenación:

```
>>> cuadrados + [36, 49, 64, 81, 100]
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

A diferencia de las cadenas de texto, que son *immutable*, las listas son un tipo *mutable*, es posible cambiar un su contenido:

```
>>> cubos = [1, 8, 27, 65, 125]  # hay algo mal aquí
>>> 4 ** 3  # el cubo de 4 es 64, no 65!
64
>>> cubos[3] = 64  # reemplazar el valor incorrecto
>>> cubos
[1, 8, 27, 64, 125]
```

También podés agregar nuevos ítems al final de la lista, usando el *método* **`append()`** (vamos a ver más sobre los métodos luego):

```
>>> cubos.append(216)  # agregar el cubo de 6
>>> cubos.append(7 ** 3)  # y el cubo de 7
```

```
>>> cubos
[1, 8, 27, 64, 125, 216, 343]
```

También es posible asignar a una rebanada, y esto incluso puede cambiar la longitud de la lista o vaciarla totalmente:

```
>>> letras = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> letras
['a', 'b', 'c', 'd', 'e', 'f', 'g']
>>> # reemplazar algunos valores
>>> letras[2:5] = ['C', 'D', 'E']
>>> letras
['a', 'b', 'C', 'D', 'E', 'f', 'g']
>>> # ahora borrarlas
>>> letras[2:5] = []
>>> letras
['a', 'b', 'f', 'g']
>>> # borrar la lista reemplzando todos los elementos por una lista vacía
>>> letras[:] = []
>>> letras
[]
```

La función predefinida **`len()`** también sirve para las listas:

```
>>> letras = ['a', 'b', 'c', 'd']
>>> len(letras)
4
```

Es posible anidar listas (crear listas que contengan otras listas), por ejemplo:

```
>>> a = ['a', 'b', 'c']
>>> n = [1, 2, 3]
>>> x = [a, n]
>>> x
[['a', 'b', 'c'], [1, 2, 3]]
>>> x[0]
['a', 'b', 'c']
>>> x[0][1]
'b'
```

Primeros pasos hacia la programación

Por supuesto, podemos usar Python para tareas más complicadas que sumar dos y dos. Por ejemplo, podemos escribir una subsecuencia inicial de la serie de *Fibonacci* así:

```
>>> # Series de Fibonacci:
... # la suma de dos elementos define el siguiente
... a, b = 0, 1
>>> while b < 10:
...     print(b)
...     a, b = b, a+b
...
1
1
2
3
5
8
```

Este ejemplo introduce varias características nuevas.

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]
```

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 (L)`
Extiende la lista agregándole todos los ítems de la lista dada. Equivale a `a[len(a):] = L`.

`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)`
Devuelve el índice en la lista del primer ítem cuyo valor sea *x*. Es un error si no existe tal ítem.

`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:

```
>>> a = [66.25, 333, 333, 1, 1234.5]
>>> print(a.count(333), a.count(66.25), a.count('x'))
2 1 0
>>> a.insert(2, -1)
>>> a.append(333)
>>> a
[66.25, 333, -1, 333, 1, 1234.5, 333]
>>> a.index(333)
1
>>> a.remove(333)
>>> a
[66.25, -1, 333, 1, 1234.5, 333]
>>> a.reverse()
>>> a
[333, 1234.5, 1, 333, -1, 66.25]
>>> a.sort()
>>> a
[-1, 1, 66.25, 333, 333, 1234.5]
>>> a.pop()
1234.5
>>> a
[-1, 1, 66.25, 333, 333]
```

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: