Listas

Josefina Sierra Santibáñez

30 de octubre de 2018

Operaciones de Secuencias

Una lista es una secuencia de elementos almacenados como un solo objeto.

Las listas de Python son un tipo de secuencia. Las siguientes operaciones son operaciones de secuencias y, por tanto, se pueden aplicar a listas.

- + concatenación
- * repetición
- <sequence>[] indexación
- <sequence>[:] sublistas (slice)
- len(<sequence>) longitud
- for <var> in <sequence> iteración sobre los elementos

También se pueden utilizar las operaciones de pertenencia

<*elem*> **in** <*list*>
<*elem*> **not in** <*list*>

Listas vs Strings

- A diferencia de las strings, que son únicamente secuencias de caracteres, las listas son secuencias de objetos arbitrarios.
- Se pueden construir listas de enteros, listas de strings, listas que contienen números y strings, e incluso listas que contienen otras listas.

```
a = [1, 2, 5, 7]
b = ["Lunes", "Martes", "Miercoles"]
c = ["Lunes", 0, "Miercoles", 8, 10]
d = ["Lunes", [0, 0], "Miercoles", [8, 10]]
```

Listas vs Strings

- Las listas son **modificables** (**mutables**). Esto significa que el valor de un elemento de una lista se puede modificar utilizando una instrucción de asignación.
- Las strings no son modificables, como muestra el siguiente ejemplo.

```
>>> a = [1, 2, 5, 7]
>>> a[2] = 0
>>> a
[1, 2, 0, 7]
>>> b = "hola"
>>> b[2] = 'j'
ERROR
```

Métodos de Listas

- El método **append** se usa para añadir un elemento al final de una lista.
- Se suele utilizar para construir una lista elemento a elemento.

```
squares = []
for x in range(1, 101):
    squares.append(x*x)
```

El fragmento de código anterior crea una lista que contiene los cuadrados de los números naturales menores o iguales que 100.

Es un patrón típico de acumulación donde la variable acumuladora es una lista.

Conversión de String a Lista

- El método **split** divide una string en una lista de substrings.
- Por defecto, divide la string siempre que encuentra uno o varios espacios en blanco.
- También puede dividir una string cuando encuentra otro **delimitador** que puede ser otra string que se proporciona como argumento. El delimitador no aparece en el resultado.

```
>>> dias = "Lunes␣Martes␣Miercoles␣Jueves"
>>> dias.split()
['Lunes', 'Martes', 'Miercoles', 'Jueves']
>>> numeros = "11,12,13,14,15"
>>> numeros.split(',')
['11', '12', '13', '14', '15']
```

Conversión de Lista a String

- El método **join** concatena los elementos de una lista de strings en una sola string utilizando la string propietaria del método como **separador** de los elementos.

```
>>> dias = ['Lunes', 'Martes', 'Miercoles',
            'Jueves']
>>> ",".join(dias)
'Lunes,Martes,Miercoles,Jueves'
>>> "--".join(dias)
'Lunes--Martes--Miercoles--Jueves'
>>> "".join(dias)
'LunesMartesMiercolesJueves'
```

Es más eficiente acumular una string con una lista

```
def capgirar_1(w):
    wi = "";
    n = len(w)
    for i in range(-1, -(n+1), -1):
        wi = wi + w[i]
    return wi
```

- capgirar_1 es ineficiente. La siguiente instrucción crea una copia de wi y añade el carácter w[i] al final de dicha copia.
  wi = wi + w[i]
- A medida que invertimos la palabra, vamos copiando cada vez una parte más larga de ella para añadir un carácter al final.
- Una forma de evitar copiar la palabra una y otra vez es usar una lista.

Es más eficiente acumular una string con una lista

- La palabra invertida se puede acumular como una lista de caracteres donde cada carácter se añade al final de la lista existente.
- Como las listas son mutables, **append** modifica la propia lista añadiendo un elemento al final, sin tener que copiar el contenido anterior de la lista a un nuevo objeto.
- Una vez que hemos acumulado todos los caracteres en el orden deseado, podemos utilizar el método *join* para concatenarlos en un solo paso.

```
def capgirar_2(w):
    li = [];
    n = len(w)
    for i in range(-1, -(n+1), -1):
        li.append(w[i])
    return "".join(li)
```

Esta es la forma estándar de acumular una string en Python.

Listas

- Las listas de Python son dinámicas, pueden crecer y encogerse durante la ejecución de un programa.
- También son heterogéneas. Es posible mezclar tipos de elementos diferentes en una sola lista.
- Como ya hemos comentado, las listas son objetos mutables.
- Es posible incluso modificar una subsecuencia de una lista mediante una instrucción de asignación a un slice (ver ejemplo).

```
>>> dias = ['Lunes', 'Martes', 'Miercoles',
            'Jueves']
>>> dias[2] = 3
>>> dias
['Lunes', 'Martes', 3, 'Jueves']
>>> dias[2:4] = ["Mie", "Jue", "Vie"]
>>> dias
['Lunes', 'Martes', 'Mie', 'Jue', 'Vie']
```

Listas

- Se puede crear una lista de elementos idénticos utilizando el operador de repetición (ver variables `contadores` y `primos`).

```
contadores = [0] * 26
# lista de 26 enteros iguales a cero

primos = [True] * 100
# lista de 100 Booleanos iguales a True

primos[0] = primos[1] = False
# vamos descartando primos

# Ver mas adelante Criba de Eratostenes
```

Listas

- Las listas se construyen normalmente elemento a elemento, utilizando el método `append`. El siguiente ejemplo almacena en una lista números enteros positivos introducidos por el usuario.
- La variable `nums` se usa como acumulador. Comienza siendo la lista vacía y en cada iteración se le añade un número al final. El bucle termina cuando se introduce un número negativo.
- Este tipo de secuencia cuya longitud no se conoce de antemano y que termina cuando aparece un elemento que cumple una condición se denomina **secuencia terminada en centinela**.

```
nums = []
x = input("Introduzca un numero: ")
while x >= 0:
    nums.append(x)
    x = input("Introduzca un numero: ")
```

Otros Métodos de Listas

El método `append` es sólo un ejemplo de los métodos de la clase `lista`. Otros métodos de dicha clase utilizados con frecuencia son

- <list>.sort() ordena la lista
- <list>.reverse() invierte la lista
- <list>.index(x) devuelve la posición de la primera ocurrencia de x
- <list>.insert(i,x) inserta x en la posición i y mueve a la derecha el resto de elementos
- <list>.count(x) devuelve el número de ocurrencias de x
- <list>.remove(x) elimina la primera ocurrencia de x
- <list>.pop(i) borra el elemento de la posición i, desplaza a la izquierda los siguientes elementos y devuelve el valor del elemento eliminado.
- <list>.extend(<newlist>) añade al final de <list> los elementos de <newlist>

Operador del

- El operador `del` permite eliminar elementos individuales o sublistas de una lista dada.
- `del` no es un método de la clase lista, sino una operación de Python que puede utilizarse con los elementos de una lista.

```
>>> a = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> del a[0]
>>> a
[2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> del a[6:]
>>> a
[2, 3, 4, 5, 6, 7]
>>> del a[2:5:2]
>>> a
[2, 3, 5, 7]
```

Valor y Referencia

- Como las strings son inmutables, Python optimiza sus recursos haciendo que dos variables que hacen referencia al mismo valor de tipo string apunten al mismo objeto.
- Se puede comprobar si dos variables apuntan al mismo objeto utilizando el operador **is**.
- Como las listas son mutables, dos listas inicializadas con el mismo valor no apuntan al mismo objeto (ver ejemplo).

```
>>> a = "banana"
>>> b = "banana"
>>> a is b
True
>>> c = [1, 2, 3]
>>> d = [1, 2, 3]
>>> c is d
False
```

Alias

- Sin embargo, como las variables apuntan a objetos, si asignamos una variable que apunta a una lista a otra variable, ambas variables apuntarán al mismo objeto (ver ejemplo).
- Este fenómeno se conoce como **alias** y consiste en que dos o más variables apuntan al mismo objeto.
- Los cambios que se realizan con un alias, afectan al valor al que apunta el otro alias.
- En general, es mejor **evitar alias** cuando trabajamos con *objetos mutables*.

```
>>> c = [1, 2, 3]
>>> d = c
>>> c is d
True
>>> c[0] = 5
>>> d
[5, 2, 3]
```

Clonación

- Si queremos modificar una lista pero también mantener una copia de la lista original, necesitamos asignar a la segunda variable una copia de la lista original en lugar de una referencia a dicha lista.
- La forma más sencilla de **clonar** una lista es utilizar el operador *slice*. Este operador crea una nueva lista.

```
>>> c = [1, 2, 3]
>>> d = c[:]
>>> c is d
False
>>> c[0] = 5
>>> d
[1, 2, 3]
```

Parámetros de tipo Lista

- Los argumentos de tipo lista se **pasan por referencia**. La función recibe una referencia a la lista, no una copia de dicha lista.
- El paso de argumentos crea, por tanto, un alias: el programa que llama a la función tiene una variable que apunta a la lista (a en el ejemplo), y la función llamada tiene un parámetro que apunta al mismo objeto de tipo lista (t en el ejemplo).
- Si una función modifica sus parámetros de tipo lista, el programa que llama a dicha función verá las modificaciones (**efectos secundarios**). Las funciones que modifican sus argumentos durante su ejecución se denominan **modificadoras**

```
def double(t):
    for i in range(len(t)):
        t[i] = 2*t[i]
if __name__ == '__main__':
    a = [2, 3, 5, 7]
    double(a)
    # En este momento a es [4, 6, 10, 14]
```

Funciones Puras

- Una función pura no produce efectos secundarios.
- Se comunica con el programa que la llama sólo a través de parámetros que no modifica y a través de su valor de retorno.
- El retorno de valores de tipo lista se realiza por referencia.
- La versión función pura de la función modificadora anterior es

```
>>> a = [2, 3, 5, 7]
>>> def double_p(t):
      d = []
      for e in t:
        d.append(2*e)
      return d
>>> c = double(a)
>>> a
[2, 3, 5, 7]
>>> c
[4, 6, 10, 14]
```

Listas Anidadas

- Una lista anidada es una lista que aparece como elemento de otra lista.
- Si queremos acceder a los elementos de una lista anidada, podemos utilizar el operador [] dos veces, teniendo en cuenta que los operadores [] se evalúan de izquierda a derecha (ver ejemplo).
- Por ejemplo, la expresión `a[2][1]` produce como resultado el segundo elemento del tercer elemento de la lista `a`.

```
>>> a = [1, 2, [5, 4, 6], 3]
>>> a[2]
[5, 4, 6]
>>> a[2][1]
4
>>> a[2][0]
5
>>> a[0]
1
```

Matrices

- Las matrices se suelen representar utilizando listas anidadas.

$$mx = \begin{pmatrix} 1 & 2 & 3 \\ 5 & 4 & 6 \\ 7 & 8 & 9 \end{pmatrix}$$

- `mx` es una lista con tres elementos, cada elemento es una fila
- la expresión `mx[f]` selecciona la fila `f` de `mx`
- el elemento situado en la fila `f` y en la columna `c` se selecciona con la expresión `mx[f][c]`
- los índices para filas y columnas comienzan en 0

```
>>> mx = [[1,2,3], [5,4,6], [7,8,9]]
>>> mx[0][2]
3
>>> mx[2][0]
7
>>> mx[1]
[5, 4, 6]
```

Más sobre listas: Leer el capítulo 11 de *How to Think Like a Computer Scientist: Learning with Python 3* de Peter Wentworth et al. (3rd edition, 2012).

Instrucción While

Un bucle indefinido se puede implementar mediante la instrucción

```
while <condition>
  <body>
```

- <condition> es una expresión Booleana
- <body> es una secuencia de una o más instrucciones
- La semántica es la siguiente: el cuerpo del bucle se ejecuta mientras la condición sea cierta. Cuando la condición es falsa el bucle termina.
- La condición se comprueba siempre antes de ejecutar el cuerpo. Si la condición es falsa inicialmente, el cuerpo no se ejecuta.

Ejemplos de uso de while

El siguiente ejemplo presenta un método iterativo de aproximación del valor de una función. La terminación del proceso iterativo depende de la precisión con que se desea calcular dicha aproximación.

En problemas similares utilizaremos una variable **epsilon** que permitirá especificar el grado de precisión de la aproximación.

Aproximación de e^x: Diseño de una función que calcule un valor aproximado para e^x.
Se sabe que e^x se puede calcular mediante la serie de Taylor

$$e^x = \sum_{i=0}^{\infty} \frac{x^i}{i!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \ldots + \frac{x^i}{i!} + \ldots$$

Observad que para i suficientemente grande los términos de esta serie son cada vez menores.

Ejemplos de uso de while

Si en lugar de sumar infinitos términos sumamos sólo los n primeros, cometemos un error. Cuantos más sumemos, menor será el error.

Denotamos mediante t el término i-ésimo de la serie y mediante s la suma de los i primeros términos de la serie. Consideraremos que la aproximación calculada es suficientemente buena cuando

$$|t| < epsilon \cdot |s|$$

```
def aprox_exp(x, epsilon):
    k = 0
    t = 1.0
    s = 1.0
    while t >= epsilon*s:
        k += 1
        t = (t/k)*x
        s += t
    return s
```

Ejemplos de uso de while

Aproximación de $cos(x)$**:** Diseño de una función que calcule un valor aproximado para $cos(x)$.
Se sabe que $cos(x)$ se puede calcular mediante la serie de Taylor

$$cos(x) = \sum_{i=0}^{\infty} (-1)^i \frac{x^{2i}}{(2i)!} = 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \ldots + \frac{x^{2i}}{(2i)!} + \ldots$$

Observad que para i suficientemente grande los términos de esta serie son cada vez menores en valor absoluto.

Denotamos mediante t el término i-ésimo de la serie y mediante s la suma de los i primeros términos de la serie. Consideraremos que la aproximación calculada es suficientemente buena cuando

$$|t| < epsilon \cdot |s|$$

Ejemplos de uso de while

```
import math

def aprox_cos(x, epsilon):
    '''
    >>> math.cos(math.pi/4)
    0.7071067811865476
    >>> aprox_cos(math.pi/4, 1e-10)
    0.7071067811865465
    '''
    k = 0
    t = 1.0
    s = 1.0
    while abs(t) >= abs(s) * epsilon:
        k += 2
        t = ( (-t) / (k*(k-1)) ) * x * x
        s += t
    return s
```