Secuencias en Python

Listas, tuplas, rangos y cadenas

Las **listas** son secuencias mutables (se pueden actualizar), que suelen utilizarse para almacenar colecciones de elementos homogéneos (donde el grado exacto de similitud variará según la aplicación).

Las **tuplas** son secuencias inmutables (no se pueden cambiar o actualizar), que suelen utilizarse para almacenar colecciones de datos heterogéneos. Las tuplas también se utilizan para casos en los que se necesita una secuencia inmutable de datos homogéneos.

El tipo ***range*** representa una secuencia inmutable de números y se utiliza comúnmente para hacer un bucle o ciclo de un número específico de repeticiones usando la estructura iterativa ***for***.

Los datos textuales en Python se manejan con objetos str, o cadenas. Las cadenas son secuencias inmutables de caracteres en código Unicode. Las cadenas literales se escriben de varias maneras:

-Comillas simples: 'permite comillas "dobles" incrustadas'

-Comillas dobles: "permite comillas 'simples' incrustadas".

-Comillas triples: '''Tres comillas simples''', """Tres comillas dobles"""

Las cadenas entre comillas triples pueden abarcar varias líneas: todos los espacios en blanco asociados se incluirán en la cadena.

OPERACIONES CON SECUENCIAS

Las operaciones de la siguiente tabla son soportadas por la mayoría de los tipos de secuencia, tanto mutables como inmutables.

Esta tabla enumera las operaciones de secuencia ordenadas en prioridad ascendente. En la tabla, s y t son secuencias del mismo tipo, n, i, j y k son enteros y x es un objeto arbitrario que cumple las restricciones de tipo y valor impuestas por s.

Las operaciones ***in*** y ***not in*** tienen las mismas prioridades que las operaciones de comparación. Las operaciones **+** (concatenación) y ***** (repetición) tienen la misma prioridad que las operaciones numéricas correspondientes.

Operación	*Resultado*	*Notas*
x in s	True si un elemento de *s* es igual a *x*, si no, False	(1)
x not in s	False si un elemento de s es igual a *x*, si no, True	(1)
s + t	Concatenación de *s* y *t*	(6)(7)
s * n o n * s	Equivalente a agregar *s* a sí mismo *n* veces	(2)(7)
s[i]	*i*-ésimo elemento de *s*, con origen en 0	(3)
s[i:j]	Subsecuencia de *s* desde el elemento en la posición *i* hasta el anterior a la posición *j*	(3)(4)

Operación	*Resultado*	*Notas*
s[i:j:k]	Subsecuencia o rebanada de *s* desde el elemento en la posición *i* hasta el anterior a la posición *j, a saltos de k posiciones*	(3)(5)
len(s)	Longitud de *s*	
min(s)	Menor elemento de *s*	
max(s)	Mayor elemento de *s*	
s.index(x[, i[, j]])	Índice posicional de la primera ocurrencia de *x* en *s* (en o desde la posición *i* y antes de la posición *j*)	(8)
s.count(x)	Cantidad total de ocurrencias de (veces que aparece) *x* en *s*	

Las secuencias del mismo tipo también admiten comparaciones. En particular, las tuplas y las listas se comparan lexicográficamente comparando los elementos correspondientes.

La sentencia de control *for* de Python recorre los elementos de cualquier secuencia (una lista, un rango o una cadena), en el orden en que aparecen en la secuencia. Por ejemplo (en el IDLE Shell de Python):

```
>>> # Medición de algunas cadenas:
... palabras = ['gato', 'municipalidad', 'psiconeuroinmunoendocrinología']
>>> for palabra in palabras:
...     print(palabra, len(palabra))
...
gato 4
municipalidades 15
psiconeuroinmunoendocrinología 30
```

Notas:

1. Mientras que las operaciones **in** y **not in** se utilizan sólo para la comprobación de contención simple en el caso general, algunas secuencias especializadas (como str) también las utilizan para la comprobación de subsecuencia:

   ```
   >>> "ue" en "huevos"
   Verdadero
   ```

2. Los valores de n menores que 0 se tratan como 0 (lo que produce una secuencia vacía del mismo tipo que s). Tenga en cuenta que los elementos de la secuencia s no se copian; se referencian varias veces. Esto a menudo confunde a los nuevos programadores de Python; considere:

   ```
   >>> listas = [[]] * 3
   >>> listas
   ```

```
[[], [], []]

>>> listas[0].append(3)

>>> listas

[[3], [3], [3]]
```

Lo que ha ocurrido es que [[]] es una lista de un solo elemento que contiene una lista vacía, por lo que los tres elementos de [[]] * 3 son referencias a esta única lista vacía. La modificación de cualquiera de los elementos de las listas modifica esta única lista. Se puede crear una lista de diferentes listas de esta manera:

```
>>> listas = [[] for i in range(3)]

>>> listas[0].append(3)

>>> listas[1].append(5)

>>> listas[2].append(7)

>>> listas

[[3], [5], [7]]
```

3. Si i o j son negativos, el índice es relativo al final de la secuencia s: se sustituye len(s) + i o len(s) + j. Pero tenga en cuenta que -0 sigue siendo 0.

4. El corte de s desde i hasta j se define como la secuencia de elementos con índice k tal que i <= k < j. Si i o j es mayor que len(s), se utiliza len(s). Si i se omite o no se usa, se usa 0. Si j se omite o no se usa, se usa len(s). Si i es mayor o igual que j, la rebanada está vacía.

5. La subsecuencia o rebanada de s de i a j con paso k se define como la secuencia de elementos con índice x = i + n*k tal que 0 <= n < (j-i)/k. En otras palabras, los índices son i, i+k, i+2*k, i+3*k y así sucesivamente, deteniéndose cuando se alcanza j (pero nunca incluyendo j). Cuando k es positivo, i y j se reducen a len(s) si son mayores. Cuando k es negativo, i y j se reducen a len(s) - 1 si son mayores. Si i o j se omiten o no se encuentran, se convierten en valores "finales" (cuyo final depende del signo de k). Tenga en cuenta que k no puede ser cero. Si k se omite, se asume 1 por defecto.

6. La concatenación de secuencias inmutables siempre da como resultado un nuevo objeto. Esto significa que la construcción de una secuencia por concatenación repetida tendrá un coste de ejecución cuadrático en la longitud total de la secuencia.

7. Algunos tipos de secuencia (como **range**) sólo admiten secuencias de elementos que siguen patrones específicos, y por tanto no admiten la concatenación o repetición de secuencias.

8. index genera un error cuando x no se encuentra en s. No todas las implementaciones admiten el paso de los argumentos adicionales i y j. Estos argumentos permiten la búsqueda eficiente de subsecciones de la secuencia. Pasar los argumentos adicionales es aproximadamente equivalente a usar s[i:j].index(x), sólo que sin copiar ningún dato y con el índice devuelto siendo relativo al inicio de la secuencia en lugar del inicio de la porción.

```
>>> 'hola'.index('j')

Traceback (most recent call last):

 File "<pyshell#8>", line 1, in <module>
```

```
'hola'.index('j')
ValueError: substring not found
```

Las operaciones de la siguiente tabla se definen en tipos de secuencia mutables (listas).

En la tabla, s es una instancia de un tipo de secuencia mutable, t es cualquier objeto iterable y x es un objeto arbitrario que cumple con las restricciones de tipo y valor impuestas por s.

Operación	*Resultado*	*Notas*
s[i] = x	El elemento en la posición *i* de *s* es reemplazado por *x*	
s[i:j] = t	La subsecuencia de *s* desde la posición *i* hasta la previa a la *j* se reemplaza por el contenido de *t*	
del s[i:j]	Equivalente a s[i:j] = []	
s[i:j:k] = t	Los elementos de la subsecuencia s[i:j:k] se reemplazan por los de *t*	(1)
del s[i:j:k]	Se eliminan los elementos de s[i:j:k] de la lista	
s.append(x)	Se agrega *x* al final de la secuencia (equivalente a s[len(s):len(s)] = [x])	
s.clear()	Se eliminan todos los elementos de *s* (equivalente a del s[:])	(5)
s.copy()	Crea una copia de *s* (equivalente a s[:])	(5)
s.extend(t) o s += t	Extiende *s* con el contenido de *t*	
s *= n	Actualiza *s* con su contenido repetido *n* veces	(6)
s.insert(i, x)	Inserta *x* en *s* en la posición *i* (equivalente a s[i:i] = [x])	
s.pop() o s.pop(i)	Devuelve el último elemento o el de la posición *i,* y también lo elimina de *s*	(2)
s.remove(x)	Elimina el primer elemento de *s* donde s[i] igual a *x*	(3)
s.reverse()	Reacomoda los elementos de *s* en orden inverso	(4)

Notas:

1. t debe tener la misma longitud que la porción que está reemplazando.
2. El argumento opcional i tiene como valor predeterminado -1, de modo que por defecto se elimina y devuelve el último elemento.

3. remove() lanza un ValueError cuando x no se encuentra en s.
4. El método reverse() modifica la secuencia en su lugar para ahorrar espacio cuando se invierte una secuencia grande. Para recordar a los usuarios que opera por efecto secundario, no devuelve la secuencia invertida.
5. clear() y copy() se incluyen por coherencia con las interfaces de los contenedores mutables que no soportan operaciones de corte (como dict y set).
6. El valor n es un entero. Los valores cero y negativos de n borran la secuencia. Los elementos de la secuencia no se copian; se referencian varias veces, como se explica para s * n en Operaciones de secuencia comunes.

Listas

Las listas son secuencias mutables, normalmente utilizadas para almacenar colecciones de elementos homogéneos (donde el grado preciso de similitud variará según la aplicación).

Las listas se pueden construir de varias maneras:

- Usando un par de corchetes para indicar la lista vacía: []
- Utilizando corchetes, separando los elementos con comas: [0], [1, 2, 3]
- Usando una definición comprensiva: [x for x in iterable]
- Usando el constructor de tipo: list() o list(iterable)

El constructor crea una lista cuyos elementos son los mismos y están en el mismo orden que los elementos de *iterable. iterable* puede ser una secuencia, un contenedor que admita la iteración o un objeto iterador. Si *iterable* ya es una lista, se realiza una copia y se devuelve, de forma similar a iterable[:]. Por ejemplo, list('abc') devuelve ['a', 'b', 'c']. Si no se proporciona ningún argumento, el constructor crea una nueva lista vacía []. list((1, 2, 3)) devuelve [1, 2, 3].

Las listas también proporcionan el siguiente método u orden posfija adicional:

sort (*** , *key = None* , *reverse = False*)

Este método ordena la lista en su lugar, utilizando únicamente comparaciones < entre elementos. No se suprimen las excepciones: si alguna operación de comparación falla, fallará toda la operación de ordenación (y es probable que la lista quede en un estado parcialmente modificado).

sort() acepta dos argumentos que sólo pueden pasarse por palabra clave (argumentos de sólo palabra clave):

key especifica una función de un argumento que se utiliza para extraer una clave de comparación de cada elemento de la lista (por ejemplo, key=str.lower). La clave correspondiente a cada elemento de la lista se calcula una vez y luego se utiliza para todo el proceso de ordenación. El valor predeterminado None significa que los elementos de la lista se ordenan directamente sin calcular un valor de clave independiente.

reverse es un valor booleano. Si se establece en True, los elementos de la lista se ordenan como si cada comparación fuera inversa (por >, es decir, ordena en forma descendente).

Este método modifica la secuencia existente para ahorrar espacio al ordenar una secuencia grande. Para recordar a los usuarios que funciona por efecto secundario, no devuelve la secuencia ordenada (se puede usar una función predefinida sorted() para solicitar explícitamente una nueva instancia de lista ordenada).

Tuplas

Las tuplas son secuencias inmutables que se utilizan normalmente para almacenar conjuntos de datos heterogéneos. Las tuplas también se utilizan en casos en los que se necesita una secuencia inmutable de datos homogéneos.

Las tuplas se pueden construir de varias maneras:

- Usando un par de paréntesis para denotar la tupla vacía: ()
- Usando una coma final para una tupla con un único elemento **'a',** o **('a',)**
- Separar elementos con comas: 'a', 'b', 'c' o ('a', 'b', 'c')
- Usando el constructor tuple() incorporado: tuple() o tuple(iterable)

El constructor crea una tupla cuyos elementos son los mismos y están en el mismo orden que los elementos de *iterable. iterable* puede ser una secuencia, un contenedor que admita la iteración o un objeto iterador. Si *iterable* ya es una tupla, se devuelve sin cambios. Por ejemplo, tuple('abc') devuelve ('a', 'b', 'c') y tuple([1, 2, 3]) devuelve (1, 2, 3). Si no se proporciona ningún argumento, el constructor crea una nueva tupla vacía ().

Tenga en cuenta que, en realidad, es la coma la que forma una tupla, no los paréntesis. Los paréntesis son opcionales, excepto en el caso de una tupla vacía o cuando son necesarios para evitar ambigüedades sintácticas. Por ejemplo, f(a, b, c) es una llamada de función con tres argumentos, mientras que f((a, b, c)) es una llamada de función con una tupla de 3 componentes como único argumento.

Las tuplas implementan todas las operaciones de secuencia comunes .

Rangos

El tipo *range* representa una secuencia inmutable de números y se utiliza comúnmente para forzar un número específico de repeticiones en bucles *for*.

La función constructora de secuencias de enteros range() se puede invocar **range**(*stop*) o **range**(*start*, *stop*[, *step*]), donde el argumento *step* es opcional.

Los argumentos del constructor de rango deben ser números enteros. Si se omite el argumento *step* (paso), el valor predeterminado es 1. Si se omite el argumento *de start* (comienzo), el valor predeterminado es 0. Si *el paso* es cero, se produce un error.

Para un *paso* positivo, el contenido de un rango r se determina mediante la fórmula r[i] = start + step*i donde i >= 0 y r[i] < stop.

Para *step* negativo, el contenido del rango todavía está determinado por la fórmula, pero las restricciones son r[i] = start + step*i, pero con i >= 0 y r[i] > stop.

Un rango resultará vacío si r[0] no cumple con la restricción de valor. Los rangos admiten índices negativos, pero estos se interpretan como indexación desde el final de la secuencia determinada por los índices positivos.

Ejemplos de rangos:

```
>>> list(range(10))

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(range(1, 11))

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

```
>>> list(range(0, 30, 5))
[0, 5, 10, 15, 20, 25]
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
>>> list(range(0, -10, -1))
[0, -1, -2, -3, -4, -5, -6, -7, -8, -9]
>>> list(range(0))
[]
>>> list(range(1, 0))
[]
```

Los rangos implementan todas las operaciones de secuencia comunes excepto la concatenación y la repetición (debido al hecho de que los rangos solo pueden representar secuencias que siguen un patrón estricto y la repetición y la concatenación generalmente violarán ese patrón).

La ventaja de este tipo range sobre un tipo list o un objeto normal tuple es que un rango siempre ocupará la misma (pequeña) cantidad de memoria, sin importar el tamaño del rango que representa (ya que solo almacena los valores start, stop y step, calculando elementos individuales y subrangos según sea necesario).

Los rangos brindan características como pruebas de contención, búsqueda de índice de elementos, segmentación y soporte para índices negativos:

```
>>> r = range(0, 20, 2)
>>> r
range(0, 20, 2)
>>> 11 in r
False
>>> 10 in r
True
>>> r.index(10)
5
>>> r[5]
10
>>> r[:5]
range(0, 10, 2)
>>> r[-1]
18
```

Dos rangos se consideran iguales si representan la misma secuencia de valores. Dos rangos que se comparan como iguales pueden tener argumentos start, stop y step diferentes, por ejemplo range(0) == range(2, 1, 3) o range(0, 3, 2) == range(0, 4, 2).

Si se necesita iterar sobre una secuencia de números, la función range() resulta muy útil, ya que genera progresiones aritméticas:

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

El punto final dado nunca es parte de la secuencia generada; range(10) genera 10 valores, los índices legales para los elementos de una secuencia de longitud 10, que son desde el 0 hasta el 9. Es posible dejar que el rango comience en otro número o especificar un incremento diferente (incluso negativo; a veces esto se llama "paso"):

```
>>> list(range(5, 10))
[5, 6, 7, 8, 9]
```

```
>>> list(range(0, 10, 3))
[0, 3, 6, 9]
```

```
>>> list(range(-10, -100, -30))
[-10, -40, -70]
```

Para iterar sobre los índices de una secuencia, se pueden combinar range() y len() de la siguiente manera:

```
>>> a = ['ta', 'te', 'ti']
>>> for i in range(len(a)):
...     print(i, a[i])
...
0 ta
1 te
2 ti
```

Sin embargo, en la mayoría de estos casos es conveniente utilizar la función enumerate():

```
>>> a = ['ta', 'te', 'ti']
>>> for i, v in enumerate(a):
...     print(i, v)
...
0 ta
1 te
2 ti
```

Ocurre algo extraño si simplemente se imprime un rango:

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

En muchos sentidos, el objeto devuelto por range() se comporta como si fuera una lista, pero en realidad no lo es. Es un objeto que devuelve los elementos sucesivos de la secuencia deseada cuando se itera sobre él, pero en realidad no crea la lista, lo que ahorra espacio.

Se dice que un objeto de este tipo es iterable, es decir, adecuado como objetivo para funciones y construcciones que esperan algo de lo que puedan obtener elementos sucesivos hasta agotar la oferta. Se ha visto que la destructura iterativa *for* es una construcción de este tipo, mientras que un ejemplo de una función que toma un iterable es sum():

```
>>> sum(range(4))  # 0 + 1 + 2 + 3
6
```

Para recorrer dos o más secuencias al mismo tiempo, las entradas se pueden emparejar con la función zip().

```
>>> preguntas = ['sobrenombre', 'equipo de fútbol', 'color favorito']
>>> respuestas = ['Lucho', 'Estudiantes de La Plata', 'azul']
>>> for p, r in zip(preguntas, respuestas):
...     print(f'¿Cuál es tu {p}?  {r}')
...
¿Cuál es tu sobrenoombre? Lucho
¿Cuál es tu equipo de fútbol?  Estudiantes de La Plata
¿Cuál es tu color favorito?  azul
```

Para recorrer una secuencia en sentido inverso, primero se especifica la secuencia en dirección hacia adelante y luego se llama a la función reversed().

```
>>> for i in reversed(range(1, 10, 2)):
...     print(i)
...
```

```
9
7
5
3
1
```

Para recorrer una secuencia ordenada, se utiliza la función sorted() que devuelve una nueva lista ordenada dejando la fuente inalterada.

```
>>> canasta = ['manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana']
>>> for fruta in sorted(canasta):
...     print(fruta)
...
banana
manzana
manzana
naranja
naranja
pera
```

El uso de la función set() en una secuencia, la convierte en un conjunto eliminando elementos duplicados. El uso de sorted() en combinación con set() en una secuencia es una forma práctica de recorrer en bucle los elementos únicos de una secuencia ordenada:

```
>>> canasta = ['manzana', 'naranja', 'manzana', 'pera', 'naranja', 'banana']
>>> for f in sorted(set(canasta)):
...     print(f)
...
banana
manzana
naranja
pera
```