Iterables en Python: Más Ejemplos

Juannie & DeepSeek

09/03/2025

Los **iterables** en Python son objetos que pueden recorrerse elemento por elemento, normalmente utilizando un bucle `for`. Esto se debe a que implementan el protocolo de iteración, lo que les permite devolver sus elementos uno a uno. Algunos ejemplos de iterables son:

- **Cadenas de texto (`str`)**: Se recorren carácter a carácter.
- **Listas (`list`)**: Secuencias ordenadas de elementos.
- **Tuplas (`tuple`)**: Secuencias ordenadas e inmutables.
- **Rangos (`range`)**: Secuencias de números.
- **Diccionarios (`dict`)**: Aunque son iterables (por defecto, iteran sobre sus claves), no son secuencias indexadas directamente.
- **Conjuntos (`set`)**: Son iterables pero no ordenados, por lo que no se les puede aplicar indexado o slicing.
- **Generadores (`generator`)**: Se generan sobre la marcha y se pueden recorrer, pero no soportan acceso por índice.

Criterio de [inicio:fin:paso] (slicing)

El **slicing** es una técnica para extraer una subsecuencia de un objeto indexado (como cadenas, listas, tuplas o rangos) usando la notación `[inicio:fin:paso]`, donde:

- **inicio**: Es el índice donde comienza el corte (incluido).
- **fin**: Es el índice donde termina el corte (excluido).
- **paso**: Es la cantidad de posiciones que se salta entre elementos.

Ejemplos con cadenas de texto:

```python
cadena = "Python"

# Extraer desde el índice 0 hasta el 3 (sin incluir el 4):
print(cadena[0:4])  # Salida: 'Pyth'

# Extraer cada 2 caracteres:
print(cadena[::2])  # Salida: 'Pto'

# Invertir la cadena:
print(cadena[::-1])  # Salida: 'nohtyP'
```

Ejemplos con listas:

```python
lista = [10, 20, 30, 40, 50, 60]

# Extraer elementos desde el índice 1 hasta el 4:
print(lista[1:5])  # Salida: [20, 30, 40, 50]
```

```
# Extraer la lista en orden inverso:
print(lista[::-1])  # Salida: [60, 50, 40, 30, 20, 10]
```

Ejemplo con tuplas:

```
tupla = (100, 200, 300, 400, 500)
print(tupla[1:4])  # Salida: (200, 300, 400)
```

Importante:
El slicing solo funciona en aquellos iterables que mantienen un orden y permiten el acceso por índice (como cadenas, listas, tuplas y rangos). En el caso de **diccionarios** o **conjuntos**, que no son secuencias indexadas, el slicing no se puede aplicar directamente. Con los **generadores**, al no almacenar todos los elementos en memoria ni tener índices, tampoco se puede usar slicing a menos que se conviertan primero a una lista.

En resumen, los iterables son estructuras de datos que se pueden recorrer y, cuando tienen un orden definido, es posible aplicarles slicing para extraer partes específicas utilizando la notación `[inicio:fin:paso]`.

Aplicación en otros iterables:

Funciona en: 1. **Listas (`list`)**
`python lista = [0, 1, 2, 3, 4, 5] print(lista[1:4]) # [1, 2, 3] print(lista[::-1]) # [5, 4, 3, 2, 1, 0] (lista invertida)`

2. **Tuplas (`tuple`)**

```
tupla = (10, 20, 30, 40)
print(tupla[:2])  # (10, 20)
```

3. **Rangos (`range`)** → Se puede convertir a lista para aplicar slicing.

```
r = range(10)
print(list(r)[2:8:2])  # [2, 4, 6]
```

No funciona directamente en: 4. **Conjuntos (`set`)**
- Son **desordenados**, por lo que no tienen índices.
- No se puede hacer slicing. `python conjunto = {1, 2, 3, 4, 5} # print(conjunto[1:4]) Error`

5. **Diccionarios (`dict`)**
   - No soportan slicing directamente.
   - Se puede aplicar en **listas de claves o valores**.

```
diccionario = {'a': 1, 'b': 2, 'c': 3}
print(list(diccionario.keys())[:2])  # ['a', 'b']
print(list(diccionario.values())[::2])  # [1, 3]
```

6. **Generadores (`generator`)**
   - No soportan slicing directamente porque **se consumen una vez y no tienen índices**.
   - Se pueden convertir en listas primero:

```
def gen():
    yield from range(10)

g = gen()
print(list(g)[2:6])  # [2, 3, 4, 5]
```

En resumen: **Cadenas, listas, tuplas y rangos soportan slicing. Sets, diccionarios y generadores no lo hacen directamente.**