Programación con

python™

Tema 4: Secuencias

Alma Mallo Casdelo - alma.mallo@udc.es

Contenedores: secuencias

- ***Una secuencia es un conjunto de datos ordenado***, donde cada dato puede ser referido por su posición, la cual se representa por un número entero no negativo.
- Las ***secuencias*** pueden ser ***inmutables***, no se pueden cambiar una vez creadas (gestión más eficiente), ***o mutables***, pueden añadirse, modificarse o quitarse elementos (utilización más flexible).
  - Inmutables: ***tuplas***, ***cadenas de caracteres***…
  - Mutables: ***listas***…
- Los elementos de una secuencia pueden ser de tipos distintos.
- Las secuencias se pueden manipular mediante:
  - Operadores: [], in, =, +, * …
  - Funciones / métodos: append(), extend(), insert(), remove() …

Ejemplos

Inicializar secuencia vacía:

```
l = []  #Lista vacía
t = ()  #Tupla vacía
c = “”  #Cadena de caracteres vacía
```

Dar un valor inicial (no vacío):

```
t = (1, 2, 3, 4)  #Tupla (inmutable)
l = [1, 2, 3, 4]  #Lista (mutable)
c = "1234"        #Cadena de caracteres
c = '1234'        #Cadena de caracteres
```

Acceder a una posición:

```
t[1]   #Resultado: 2
l[1]   #Resultado: 2
c[1]   #Resultado: ‘2'
```

Acceder a un rango de elementos:

```
t[1:3]  #Resultado: (2, 3)
l[1:3]  #Resultado: [2, 3]
c[1:3]  #Resultado: ’23’
```

Concatenar:

```
t2 = t + t  #Resultado: (1,2,3,4,1,2,3,4)
l2 = l + l  #Resultado: [1,2,3,4,1,2,3,4]
c2 = c + c  #Resultado: ’12341234'
```

Repetir:

```
ceros = [0] * 3  #Resultado: [0, 0, 0]
t2 = t * 3       #Resultado: (1,2,3,4,1,2,3,4,1,2,3,4)
l2 = l * 3       #Resultado: [1,2,3,4,1,2,3,4,1,2,3,4]
c2 = c * 3       #Resultado: ’123412341234'
```

Longitud (número de elementos):

```
len(l)  #Resultado: 4
len(t)  #Resultado: 4
len(c)  #Resultado: 4
```

Listas: métodos

- list.append(x): Añade un elemento al final de la lista.
- list.extend(iterable): Amplía la lista añadiendo todos los elementos del iterable.
- list.insert(i, x): Inserta un elemento en una posición dada. El primer argumento es el índice del elemento anterior al que se va a insertar, por lo que a.insert(0, x) inserta al principio de la lista, y a.insert(len(a), x) equivale a a.append(x).
- list.remove(x): Elimina el primer elemento de la lista cuyo valor es igual a x. Genera un error ValueError si no existe tal elemento.
- list.pop([i]): Elimina el elemento en la posición dada de la lista y lo devuelve. Si no se especifica ningún índice, a.pop() elimina y devuelve el último elemento de la lista. El parámetro “i” es opcional (por eso está entre corchetes, no se escriben los corches al usar el método).

Listas: métodos

- list.clear(): Elimina todos los elementos de la lista.
- list.index(x[, start[, end]]): Devuelve el índice en la lista del primer elemento cuyo valor es igual a x. Genera un error ValueError si no existe tal elemento. Start y end: opcionales, se interpretan como en la notación slice y se utilizan para limitar la búsqueda a una subsecuencia concreta de la lista. El índice devuelto se calcula en relación con el principio de la secuencia completa en lugar del argumento start.
- list.count(x): Devuelve el número de veces que x aparece en la lista.
- list.sort(*, key=None, reverse=False): Ordena los elemento de la lista (modifica la lista). Key y reverse se pueden utilizar para personalizar la ordenación.
- list.reverse(): Da la vuelta a la lista (modifica la lista)
- list.copy(): Devuelve una copia de la lista. Equivalente a a[:].

Ejemplos (con listas)

Empezamos con la lista siguiente:

```
l = [1, 2, 3, 4]
```

Asignamos valores:

```
l[1] = -1          #Resultado: [1, -1, 3, 4]
l[1:3] = [0, 0]    #Resultado: [1, 0, 0, 4]
```

Añadimos un elemento al final:

```
l.append(8)        #Resultado:[1, 0, 0, 4, 8]
```

Añadimos el contenido de otra lista:

```
l.extend([4, 3, 2, 1])
#Resultado: [1, 0, 0, 4, 8, 4, 3, 2, 1]
```

Insertamos un valor en una posición:

```
l.insert(3, 55)  #Insertar en posición 3
#Resultado: [1, 0, 0, 55, 4, 8, 4, 3, 2, 1]
```

Borramos (por contenido):

```
#Borrar primera ocurrencia del valor 8
l.remove(8)
#Resultado: [1, 0, 0, 55, 4, 4, 3, 2, 1]
```

Borramos (por posición):

```
#Borrar el elemento de la posición 6
del l[6]
#Resultado: [1, 0, 0, 55, 4, 4, 2, 1]
```

Ejemplos (con listas)

Copiar (mal):

```
l = [1, 0, 0, 55, 4, 4, 2, 1]

l2 = l  #No es una copia, son la misma lista!

l2 is l  #True

l2[0] = 33  #Implica que l[0] también es 33
#Contenido de l:  [33, 0, 0, 55, 4, 4, 2, 1]
#Contenido de l2: [33, 0, 0, 55, 4, 4, 2, 1]
```

Copiar (bien):

```
l = [1, 0, 0, 55, 4, 4, 2, 1]

l2 = l[:]  #l2 es una nueva lista, copia de l

l2 is l  #False

l2[0] = 33
#Contenido de l:  [1, 0, 0, 55, 4, 4, 2, 1]
#Contenido de l2: [33, 0, 0, 55, 4, 4, 2, 1]
```

Recorrer todos los elementos con **for** (solo hacen falta los valores):

```
for i in l:
    print(i, end=" ")
```

Recorrer todos los elementos con **for** (hacen falta índices y valores):

```
for pos, value in enumerate(l):
    print("Posición", pos, ", valor", value)
```

Recorrer todos los elementos con **while**:

```
i = 0
while i < len(l):
    print(l[i], end=" ")
    i = i + 1
```

Listas por comprensión

- Forma compacta de generar listas a partir de sentencias iterativas `for`.
- Consisten en corchetes que contienen una expresión seguida por una sentencia `for` y, opcionalmente, más cláusulas `for` o `if` adicionales.
- Ejemplo con un único `for` y un único `if`:
  - `[expresión for variable in iterador if condición]`
- El resultado es una nueva lista formada por los elementos resultado de evaluar `expresión` para cada iteración de `for`.

Generadores

- Las listas por comprensión pueden necesitar mucha memoria.
- Los generadores tienen una sintaxis prácticamente igual (simplemente cambian los corchetes por paréntesis). Ejemplo:
  - `(expresión for variable in iterador if condición)`
- En vez de crear una lista, crean un iterador que calcula un elemento nuevo cada vez que lo pedimos.
- Lo anterior implica que ***un generador solo se puede usar una vez***, no podemos utilizarlo para acceder a un mismo elemento varias veces.

Ejemplos

```
In [1]: lista = [x for x in range(10)]

In [2]: lista
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [3]: lista = [x**2 for x in lista]

In [4]: lista
Out[4]: [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

In [5]: lista = [x for x in lista if x > 25]

In [6]: lista
Out[6]: [36, 49, 64, 81]

In [7]: lista2 = [49, 81, 120, 180]

In [8]: lista3 = [x for x in lista for y in lista2 if x == y]

In [9]: lista3
Out[9]: [49, 81]
```

```
In [10]: falsa_lista = (x for x in range(10))

In [11]: type(lista)
Out[11]: list

In [12]: type(falsa_lista)
Out[12]: generator

In [13]: for i in falsa_lista:
    ...:     print(i, end=" ")
    ...:
0 1 2 3 4 5 6 7 8 9
In [14]: for i in falsa_lista:
    ...:     print(i, end=" ")
    ...:

In [15]:
```

No hay salida por pantalla porque el iterador ya recorrió todos los elementos en el bucle anterior.

Strings: métodos

- str.join(iterable) => devuelve una cadena de caracteres resultado de unir las cadenas de caracteres que proporciona el objeto iterable intercalando el propio valor de str entre ellas. Ejemplo:

```
“;”.join([“abc”, “def”, “ghi”])  #Resultado: “abc;def;ghi”
```

- str.split(separador) => devuelve una lista de cadenas de caracteres resultado de dividir el valor de str utilizando la cadena especificada en separador. Ejemplo:

```
“abc;def;ghi”.split(“;”)  #Resultado: [“abc”,“def”,“ghi”]
```

String: métodos

- str.capitalize(): Devuelve una copia de la cadena con su primer carácter en mayúsculas y el resto en minúsculas.

- str.count(sub[, start[, end]]): Devuelve el número de ocurrencias de la subcadena sub en el rango [inicio, fin]. Los argumentos opcionales start y end se interpretan como en la notación slice. Si sub está vacía, devuelve el número de cadenas vacías entre caracteres que es la longitud de la cadena más uno.

- str.find(sub[, start[, end]]): Devuelve el índice más bajo de la cadena donde se encuentra la subcadena sub dentro del segmento s[inicio:fin]. Los argumentos opcionales start y end se interpretan como en la notación slice. Devuelve -1 si no se encuentra sub.

- str.isnumeric(): Devuelve True si todos los caracteres de la cadena son numéricos y hay al menos un carácter, False en caso contrario.

String: métodos

- str.lower(): Devuelve una copia de la cadena con todos los caracteres convertidos a minúsculas.
- str.upper(): Devuelve una copia de la cadena con todos los caracteres convertidos a mayúsculas.
- str.strip([chars]): Devuelve una copia de la cadena con los caracteres iniciales y finales eliminados. El argumento chars es una cadena que especifica el conjunto de caracteres que deben eliminarse. Si se omite o es None, el argumento chars elimina por defecto los espacios en blanco.
- str.lstrip([chars]): Igual que strip pero solo elimina los caracteres iniciales.
- str.rstrip([chars]): Igual que strip pero solo elimina los caracteres finales.
- str.replace(old, new[, count]): Devuelve una copia de la cadena con todas las apariciones de la subcadena old sustituidas por new. Si se proporciona el argumento opcional count, sólo se sustituyen las primeras count ocurrencias.

Ejemplo con cadenas

- El siguiente ejemplo devuelve las minúsculas y las mayúsculas que hay en la variable cadena:

```
minusculas = ""
mayusculas = ""
cadena = "Hola Mundo"
for letra in cadena:
    if letra.islower():
        minusculas += letra
    elif letra.isupper():
        mayusculas += letra
print(minusculas)
print(mayusculas)
```

Resultado de ejecución:

```
olaundo
HM
```

Referencias

- https://docs.python.org/3/library/stdtypes.html#sequence-types-list-tuple-range
- https://docs.python.org/3/tutorial/datastructures.html#list-comprehensions
- https://docs.python.org/3/library/stdtypes.html#text-sequence-type-str
- https://docs.python.org/3/library/stdtypes.html#string-methods