

Recursos con listas

Unidad 4

Apunte de cátedra
2do Cuatrimestre 2023

Pensamiento computacional (90)
Cátedra: Camejo

.UBA XXI

1. Recursos con listas

Es importante recordar que las listas, las strings, los rangos y las tuplas son un tipo de secuencia, por lo tanto, todas las operaciones, funciones, métodos y recursos en general, que existen para las secuencias, se pueden usar para esas estructuras.

Esto es porque Python está implementado, usando un enfoque de programación que se conoce como **Orientado a Objetos (OO)**, en el que todo lo que se define para un tipo, también se aplica para sus subtipos. Esto quiere decir que, como las listas, strings, rangos y tuplas son tipos de secuencias, comparten las funciones que se pueden aplicar y los recursos de las secuencias. Esta es la propiedad de **herencia** del enfoque orientado a objetos, que es lo que habilita esta manera de compartir recursos. Funciona como las pertenencias en un hogar cualquiera: la ropa, los dispositivos, el dinero de los padres es considerado también propiedad de los hijos, pero esto no ocurre a la inversa.

Así que todas las operaciones, funciones, métodos, recursos en general, que definamos para el supertipo de las tuplas, strings y listas (secuencias) es heredado por ellas. Claro que a su vez, cada una de estas estructuras tiene sus peculiaridades. Por eso siempre hay recursos extra, propios de esos subtipos en particular.

En este apunte vamos a estar trabajando con el subtipo de secuencia llamado Lista. Acá tenemos un listado de los principales métodos que se emplean para trabajar con lista (algunas de las cuales ya las vimos en el apunte anterior:

Método	Definición	Ejemplo
append(valor)	Agrega el elemento valor al final de la lista.	<pre>lista = [2, 3, 5] lista.append(1) # la lista queda: [2, 3, 5, 1]</pre>
insert(posición, valor)	Inserta el elemento valor en la posición posición.	<pre>lista = ['h', 'l', 'a'] lista.insert(1, 'o') # la lista queda: ['h', 'o', 'l', 'a']</pre>
remove(valor)	Quita de la lista el elemento valor.	<pre>lista = [1, 3, 5] lista.remove(3) # la lista queda: [1, 5]</pre>
pop([índice])	Quita de la lista el elemento de la	<pre>lista = ['d', 'i', 'a', 'a']</pre>

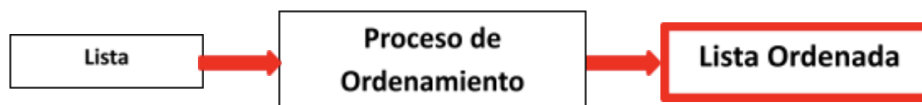
	posición índice. Si no se usa este parámetro, se quita el último elemento.	<pre> valor = lista.pop() # la lista queda: ['d', 'i', 'a'] y en la variable valor queda guardado el elemento que sacamos (en este caso 'a') </pre>
extend(otra_lista)	Agrega al final de la lista otra_lista	<pre> lista = [1, 2, 2] otra_lista = [4, 7] lista.extend(otra_lista) # la lista queda: [1, 2, 2, 4, 7] </pre>
sort([reverse=True], [key=función])	Ordena la lista. Si se emplea el parámetro reverse, en orden descendente, si se usa key, con criterio de ordenamiento función.	<pre> lista = [1, 4, 3, 2] lista.sort() # la lista queda: [1, 2, 3, 4] </pre>
reverse()	Invierte el orden de la lista (el primero pasa a ser último)	<pre> lista = ['h', 'o', 'l', 'a'] lista.reverse() # la lista queda: ['a', 'l', 'o', 'h'] </pre>
count(valor)	Cuenta la cantidad de apariciones de valor en la lista.	<pre> lista = [4, 1, 2, 5, 1] cantidad = lista.count(1) # la lista queda igual y cantidad vale 2 porque el 1 aparece dos veces </pre>
index(valor)	Devuelve la posición de la primera aparición de valor en la lista.	<pre> lista = [1, 4, 3, 2] lugar = lista.index(4) # la lista queda igual y lugar vale 1 porque el 4 está en la posición 1 </pre>

Con ellos podemos resolver muchas de las necesidades que tengamos a la hora de trabajar almacenando nuestros datos en una lista. Algunos son simples de aplicar y no tienen demasiadas vueltas. Otros son más flexibles y permiten una interesante variedad de usos.

Ahora vamos a concentrarnos en el empleo del método `sort()`, que se usa para ordenar listas (y de su prima hermana, la función `sorted()`, que hace lo propio para todo tipo de secuencia, incluidas las listas).

1.1. Ordenamiento de Estructuras

Ordenar una Estructura de Datos (como una lista en Python) requiere un algoritmo especial que permita cambiar los elementos de lugar de modo que al final la Estructura contenga los mismos elementos, pero acomodados probablemente en otras posiciones, de acuerdo a ese orden.



Es importante saber que no importa si la Estructura estaba ordenada total, parcialmente o nada de antemano; el resultado debe ser el mismo. Es decir, si vamos a ordenar, para cualquier estado inicial de la lista, el estado final siempre tiene que ser el mismo.

Y además, **toda estructura que se someta a ordenamiento se ordena aunque ya esté ordenada**. Esto es porque el esfuerzo de comprobar antes (y en la mayoría de los casos ordenar después) es superior a ordenar todo sin preguntar, ya que el ordenamiento innecesario sólo se acabará dando en una minoría de ellas.

Un buen algoritmo de ordenamiento debe garantizar devolver la Estructura ordenada, en caso de que no lo haya estado, y no debe desordenarla si ya estaba ordenada previamente.

Existen una infinidad de Algoritmos de Ordenamiento y están a un click de distancia si deseamos aplicar alguno específico. Sin embargo, Python ya tiene empaquetada la funcionalidad de ordenamiento en función o método. Y debemos decir que son suficientemente flexibles como para que apliquemos directamente esos recursos cuando necesitemos ordenar.

Trabajar con datos ordenados tiene muchos beneficios. Pero **los Algoritmos de Ordenamiento requieren bastante esfuerzo computacional**. Esfuerzo que crece, a veces exponencialmente, a medida que trabajamos con listas más grandes. Con esto no pretendemos decir que no hay que ordenar cuando consideremos necesario, ni mucho menos. Pero debemos ser conscientes de que implica un esfuerzo computacional muchas veces importante.

Vamos a analizar las opciones que tenemos en Python para ordenar secuencias. Podemos usar la función `sorted()`, o el método `sort()`, sólo para listas. El empleo de ambos es muy similar; sin embargo, el método `sort()` modifica la lista sobre la que se aplica el ordenamiento y no devuelve nada (`NONE`), mientras que `sorted()` no toca la Estructura y devuelve una versión ordenada (por eso se puede aplicar a tuplas o strings).

Miremos cómo emplear correctamente cada una de ellas:

Método `sort()`

```
lista = [4, 1, 2, 7]
```

```
lista.sort()
```

la **lista** queda [1, 2, 4, 7]

Función sorted()

```
lista = [4, 1, 2, 7]
```

```
lista_ordenada = sorted(lista)
```

la lista queda igual, y se guarda en **lista_ordenada** la lista ordenada [1, 2, 4, 7]

Nota

Recordemos que cuando pasamos una lista a una función, entregamos el permiso de acceso a la lista original. Por eso, si pasamos una lista a una función, y esta la modifica, ordenándola como en el ejemplo, cuando la función finaliza, la lista queda ordenada en el programa que invocó a la función. En cambio, si obtenemos una copia de la lista devuelta por la función `sorted()`, esa copia es local a la función y debe ser devuelta con `return` para que resulte visible afuera.

Un par de ejemplos de cómo usar la función `sorted()` con tuplas o strings:

Programa	Salida
<pre>t=(10,2,-3.5,0) print('tupla inicial:') print(t) print('tupla ordenada como lista:') l=sorted(t) print(l) t=tuple(l) print('tupla ordenada como tupla:') print(t)</pre> <pre>s='Abracadabra' print('string inicial:') print(s) print('string ordenada como lista:') l=sorted(s) print(l) s=''.join(l) print('string ordenada como string:') print(s)</pre>	<pre>tupla inicial: (10, 2, -3.5, 0) tupla ordenada como lista: [-3.5, 0, 2, 10] tupla ordenada como tupla: (-3.5, 0, 2, 10)</pre> <pre>string inicial: Abracadabra string ordenada como lista: ['A','a','a','a','a','b','b','c','d','r','r'] string ordenada como string: Aaaaabbcdrr >>></pre>

Nota

Independientemente del tipo de secuencia que enviemos, `sorted()` devolverá siempre una lista.

1.1.1. Elementos Comparables

Todos los elementos de la secuencia a ordenar deben ser comparables; es decir, deben tener tipos compatibles para la comparación (números con números, string con string, etc.). Si no se pueden comparar entre si, no hay forma de decidir cuál es menor o mayor.

Cuando la comparación se hace entre dos secuencias (string, tupla, lista) se compara elemento a elemento, de izquierda a derecha, hasta desempatar, que alguna finalice, o determinar que son iguales.

Por ejemplo: `lista=[1, 'camila', True, 2]` **no** es ordenable.

Veamos un par de ejemplos de cómo usar la función **`sorted()`** con tuplas y strings:

Tuplas

main.py

```
1 t=(10,2,-3.5,0)
2 print('tupla inicial:')
3 print(t)
4 print('tupla ordenada como lista:')
5 l=sorted(t)
6 print(l)
7 t=tuple(l)
8 print('tupla ordenada como tupla:')
9 print(t)
10
```

```
tupla inicial:
(10, 2, -3.5, 0)
tupla ordenada como lista:
[-3.5, 0, 2, 10]
tupla ordenada como tupla:
(-3.5, 0, 2, 10)
```

Strings

main.py

```
1 s='Abracadabra'
2 print('string inicial:')
3 print(s)
4 print('string ordenada como lista:')
5 l=sorted(s)
6 print(l)
7 s=''.join(l)
8 print('string ordenada como string:')
9 print(s)
10
```

```
string inicial:
Abracadabra
string ordenada como lista:
['A', 'a', 'a', 'a', 'a', 'b', 'b', 'c', 'd', 'r', 'r']
string ordenada como string:
Aaaaabbcdrr
```

⚠ ¡Atención!

Independientemente del tipo de secuencia que enviemos, `sorted()` devolverá siempre una lista.

Tanto `sort()` como `sorted()` disponen de un argumento opcional: **reverse**, que por defecto está seteado en **False**. Si deseamos ordenar una lista o secuencia en forma **descendente** bastará con prender en **True** este parámetro.

Método <code>sort()</code>	Función <code>sorted()</code>
<pre>lista = [10,22,-3.5,0] print('lista inicial:') print(lista) print('lista ordenada descendente:') lista.sort(reverse=True) print(lista)</pre> <p>Salida: lista inicial: [10, 22, -3.5, 0] lista ordenada descendente: [22, 10, 0, -3.5] >>></p>	<pre>lista = [10,22,-3.5,0] print('lista inicial:') print(lista) print('lista ordenada descendente:') lista = sorted(lista,reverse=True) print(lista)</pre> <p>Salida: lista inicial: [10, 22, -3.5, 0] lista ordenada descendente: [22, 10, 0, -3.5] >>></p>

⚠ ¡Atención!

No es lo mismo el método `sort()` con el parámetro `reverse=True` que el método `reverse()`. Este último sólo invierte el orden en el que está la lista sin ordenarla.

Claro que las líneas de programa

```
lista.sort(reverse=True)
```

y

```
lista.sort()
lista.reverse()
```

producen el mismo resultado.

En cambio:

```
lista.reverse()
lista.sort()
```

No producen el mismo resultado.

Así como hay un método **reverse()** para listas, hay una función **reversed()** que se puede aplicar a secuencias. **reversed()** devuelve la secuencia en orden invertido (no ordenada). **Para usarla correctamente, debemos castear el resultado a lista o tupla.**

Va un ejemplo:

```
main.py
1 l = [1,2,3,4,5,6]
2 t = ('a','b','c')
3
4 print('l =',l)
5 print('t =',t)
6
7 a = list(reversed(l))
8 print('l inversa como lista =',a)
9
10 b = tuple(reversed(l))
11 print('l inversa como tupla =',b)
12
13 a = list(reversed(t))
14 print('t inversa como lista =',a)
15
16 b = tuple(reversed(t))
17 print('t inversa como tupla =',b)
18
19 txt = 'hola'
20 print(txt)
21
22 palabra = list(txt)
23 palabra = list(reversed(palabra))
24
25 txt = ''.join(palabra)
26 print('invertido=',txt)
27
```

```
l = [1, 2, 3, 4, 5, 6]
t = ('a', 'b', 'c')
l inversa como lista = [6, 5, 4, 3, 2, 1]
l inversa como tupla = (6, 5, 4, 3, 2, 1)
t inversa como lista = ['c', 'b', 'a']
t inversa como tupla = ('c', 'b', 'a')
hola
invertido= aloh
```

1.1.2. Usando Criterios de Ordenamiento no Estándar

Miremos este programa de ordenamiento de una lista de nombres (strings):

Programa	Salida
<pre>lista=['Juan','ana','sergio','ELEna','ELEONORA','anALía'] print('Lista Inicial') for n in lista: print (n) lista.sort() print() print('Lista Ordenada')</pre>	<pre>Lista Inicial Juan ana sergio ELEna ELEONORA anALía Lista Ordenada ELEONORA</pre>

<pre>for n in lista: print (n)</pre>	ELEna Juan anALía ana sergio
--	--

¿Qué pasó acá? Como podemos ver, la lista no quedó ordenada alfabéticamente.

Podemos explicar esta salida porque las comparaciones de caracteres se hacen empleando su **código Unicode** y el código de las letras mayúsculas es menor al de las minúsculas. De modo que para resolver el dilema debemos hacer una comparación entre objetos similares.

Veamos cómo podemos realizar esto:


Programa	Salida
<pre>lista = ['Juan','ana','sergio','ELEna','ELEONORA','anALía'] print('Lista Inicial') for n in lista: print (n) for n in range(len(lista)): lista[n] = lista[n].lower() lista.sort() print() print('Lista Ordenada') for n in lista: print (n)</pre>	Lista Inicial Juan ana sergio ELEna ELEONORA anALía Lista Ordenada ana analía elena eleonora juan sergio

Acá obtendremos un correcto ordenamiento alfabético. Sin embargo, hemos alterado la forma original de las cadenas en la lista. ¿Qué pasaría si no quisiéramos eso? Deberíamos usar el parámetro **key** del método **sort()** (o la función **sorted()**). Éste permite indicar una función a aplicarse a cada elemento y emplear como **criterio de comparación para el ordenamiento** su resultado, sin que se modifiquen los datos.

Con este cambio, el programa quedaría así:

Programa	Salida
<pre>lista=['Juan','ana','sergio','ELEna','ELEONORA','anALía'] print('Lista Inicial') for n in lista: print (n)</pre>	Lista Inicial Juan ana sergio ELEna ELEONORA

<pre>lista.sort(key=str.lower) print() print('Lista Ordenada') for n in lista: print (n)</pre>	<p>anALía</p> <p>Lista Ordenada</p> <p>ana</p> <p>anALía</p> <p>ELEna</p> <p>ELEONORA</p> <p>Juan</p> <p>sergio</p>
---	---

 **Nota:** Como **lower()** no es una función, sino un *método*, siempre se invoca vinculado a un dato o tipo de dato. Por eso se usa **key=str.lower**

Si quisiéramos ordenar la misma lista de nombres pero por longitud de nombres podríamos usar la función **len()** en el parámetro **key**. Observemos:

Programa	Salida
<pre>lista=['Juan','ana','sergio','ELEna','ELEONORA','anALía'] print('Lista Inicial') for n in lista: print (n) lista.sort(key=len) print() print('Lista Ordenada') for n in lista: print (n)</pre>	<p>Lista Inicial</p> <p>Juan</p> <p>ana</p> <p>sergio</p> <p>ELEna</p> <p>ELEONORA</p> <p>anALía</p> <p>Lista Ordenada</p> <p>ana</p> <p>Juan</p> <p>ELEna</p> <p>sergio</p> <p>anALía</p> <p>ELEONORA</p>

¡Atención!

Solo escribimos el nombre de la función o método, no van ni los paréntesis ni indicamos los argumentos. Es la misma función **sorted()** o el método **sort()** los que administrarán esas invocaciones.

¿Qué pasa si no encontramos una función o método predefinido que nos sirva para el criterio de ordenamiento que deseamos establecer? En Ingeniería de Software siempre pensamos que “*Si no existe, se inventa*”. Así que, siempre podemos definir una función propia a tales efectos.

Por ejemplo, si quisiéramos ordenar la lista de nombres anteriores en orden descendente por cantidad de vocales en cada nombre:

Programa	Salida
<pre>def cant_vocales(palabra): palabra = palabra.lower() cant = 0 for v in ('a','e','i','o','u','á','é','í','ó','ú'): cant += palabra.count(v) return cant lista = ['Juan','ana','sergio','ELEna','ELEONORA','anALía'] print('Lista Inicial') for n in lista: print (n) lista.sort(reverse=True, key=cant_vocales) print() print('Lista Ordenada') for n in lista: print (n)</pre>	<p>Lista Inicial</p> <p>Juan ana sergio ELEna ELEONORA anALía</p> <p>Lista Ordenada</p> <p>ELEONORA anALía sergio ELEna Juan ana</p>

⚠ ¡Atención!

La función propia que definamos para criterio de ordenamiento (`cant_vocales`, en el ejemplo) debe recibir un parámetro del **tipo de los elementos de la Estructura a ordenar**. Y **no puede recibir absolutamente nada más desde el exterior**. Eso es porque será el propio método `sort()` (o la función `sorted()`) quien le pase a la función (`cant_vocales`, en este caso) el argumento cada vez que compare elementos. Y **sólo le enviará un elemento de la Estructura por vez**.

1.2. Map y filter

Existen dos funciones predefinidas muy útiles para manejar secuencias: `map()` y `filter()`

1.2.1. Map()


`map()` aplica una función cualquiera a cada elemento de una secuencia y genera otra secuencia con los resultados.

Sintaxis de map:

`map(función, secuencia)`

Recibe la función que aplica a cada elemento de la secuencia que recibe como segundo parámetro. Por ejemplo, podemos elevar al cuadrado cada elemento de la lista:

Ejemplo	Salida
<pre>def cuadrado(numero): return numero * numero lista = [3, 6, 1, 2] print("Lista: ", lista) nueva_lista = list(map(cuadrado, lista)) print("Nueva lista: ", nueva_lista)</pre>	<pre>Lista: [3, 6, 1, 2] Nueva lista: [9, 36, 1, 4]</pre>

 **Nota:** En el ejemplo transformamos lo que nos devuelve map a una lista, usando la función `list()` para que en la variable `nueva_lista` quede guardado el resultado como una lista y mostrarlo por pantalla. Esto es porque tanto map como filter devuelven algo raro, no una lista, sino así como un mapa de recorrido, con la lista de direcciones que se deben visitar.

Sin embargo, para recorrer lo que nos devuelve map() con un for, por ejemplo, no es necesario transformarlo a lista. Es decir en el ejemplo anterior podríamos haber hecho lo siguiente para mostrar los elementos:

```
nueva_lista = map(cuadrado, lista)
for n in nueva_lista:
    print(n)
```

1.2.2. Filter()

filter() selecciona elementos de una secuencia de acuerdo a la evaluación hecha por una función booleana (función que devuelve sólo **True** o **False**) y entrega la secuencia con los elementos que pasaron el filtro (aquellos para los que la función devolvió True)

Sintaxis de filter:

filter(función,secuencia)

Recibe una función que evalúa cada elemento de la secuencia que recibe como segundo parámetro, y si la función devuelve True, entonces ese elemento queda en la lista.

Por ejemplo, si quisiéramos quedarnos con las palabras que empiezan con 'm' de una lista:

Ejemplo	Salida
---------	--------

<pre>def empieza_con_m(palabra): return palabra[0] == 'm' lista = ["manzana", "balde", "molde", "marrón", "carro", "pera"] print("Lista: ") for i in lista: print(i) print() nueva_lista = list(filter(empieza_con_m, lista)) print("Nueva lista: ") for n in nueva_lista: print(n)</pre>	<p>Lista:</p> <p>manzana balde molde marrón carro pera</p> <p>Nueva lista:</p> <p>manzana molde marrón</p>
--	--

Como ya dijimos anteriormente, `filter` devuelve algo raro, por lo que también lo casteamos (castear es transformar un tipo de dato a otro) a lista con `list()` para guardar el resultado como lista en la variable `nueva_lista`.

Otro ejemplo:

En el siguiente ejemplo partimos de una lista de números naturales y obtenemos una lista con los cuadrados y otra sólo con los pares. Nada de recorridos de la lista original para lograrlo. Todo el bucle queda incluido en la función. Por eso se llaman funciones comprimidas.

Programa	Salida
<pre>def cuadrado(x): return x ** 2 def par(x): if x%2==0: return True else: return False lista=[1,2,3,4,5,6] print('Lista Original') print(lista) lis = list(map(cuadrado,lista)) print('Lista de cuadrados') print(lis) filtro=list(filter(par,lista)) print('Lista de Pares') print(filtro)</pre>	<p>Lista Original [1, 2, 3, 4, 5, 6]</p> <p>Lista de cuadrados [1, 4, 9, 16, 25, 36]</p> <p>Lista de Pares [2, 4, 6] >>></p>

Bibliografía Adicional

Wachenchauzer, R. (2018). *Aprendiendo a programar usando Python como herramienta*. Capítulo 7.