Recursos con listas

Unidad 4

Apunte de cátedra

Pensamiento computacional (90)
Cátedra: Camejo

.UBAXXI

Recursos con listas

Lo primero que debemos recordar es que las **listas**, las **strings** y las **tuplas** son un tipo de secuencia, por lo tanto (y debido a que Python está implementado usando un enfoque de programación que se conoce como **Orientado a Objetos** (OO)), todo lo de un tipo es también de sus subtipos. Funciona como las pertenencias en un hogar cualquiera: la ropa, los dispositivos, el dinero de los padres es considerado también de los hijos, pero esto no ocurre a la inversa! Esta misma lógica es la que se aplica en la **Propiedad de Herencia** de un Enfoque OO, que es lo que habilita esta manera de compartir recursos.

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.

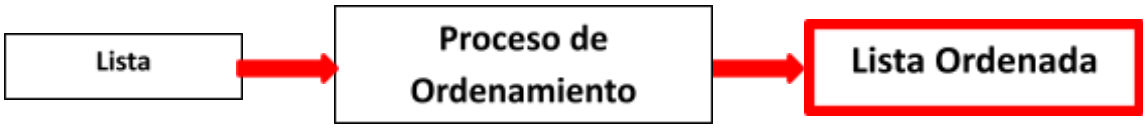
Acá tenemos una lista con los principales métodos que se emplean para trabajar con listas:

lista.**append**(valor) → Agrega el elemento valor al final de la lista.
lista.**insert**(posic,valor) → Inserta el elemento valor en la posición posic.
lista.**remove**(valor) → Quita de la lista el elemento valor.
lista.**pop**([índice]) → Quita de la lista el elemento de la posición índice. Si no se usa este parámetro, quita el último elemento.
lista.**extend**(otraLista) → Agrega al final de lista otraLista.
lista.**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.
lista.**reverse**() → Invierte el orden de la lista (el primero pasa a ser último)
lista.**count**(valor) → Cuenta la cantidad de apariciones de valor en la lista.
lista.**index**(valor) → Devuelve la posición de la primera aparición de valor en la lista.

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. Nos concentraremos ahora 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, incluídas las listas).

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 (lista, por ejemplo) 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.

Y además, toda Estructura se somete a ordenamiento, aunque ya esté ordenada, porque el esfuerzo de comprobar antes, y en la mayoría de los casos ordenar después, es superior a ordenar todo sin preguntar. Considerando 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, como sucede con muchos otros algoritmos básicos y muy utilizados (**min, abs, sum,** etc.), 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. **Nada mejor que tener a los mejores programadores del planeta trabajando para nosotros!**

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 combinando con funciones propias:

Método sort()	**Función sorted()**

`def ordena(l):`
`  l.sort()`
 	
`lista1=[10,2,-3.5,0]`
`ordena(lista1)`
`print('Lista1:')`
`print(lista1)`
	`def ordena(l):`
	`  lista2=sorted(l)`
	`  return lista2`
	
	`lista1=[10,2,-3.5,0]`
	`lista1=ordena(lista1)`
	`print('Lista1:')`
	`print(lista1)`

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 list**a 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**

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

Para tener en cuenta:

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

Importante!

Todos los elementos de la secuencia a ordenar deben ser **comparables**; es decir, deben tener tipos compatibles para 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:

`lis=[1,'camila',True,2]` **no** es ordenable.

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 sort()	**Función sorted()**

`l=[10,22,-3.5,0]`
`print('lista inicial:')`
`print(l)`
`print('lista ordenada descendente:')`
`l.sort(reverse=True)`
`print(l)`
 	
Salida:
`lista inicial:`
`[10, 22, -3.5, 0]`
`lista ordenada descendente:`
`[22, 10, 0, -3.5]`
`>>>`
	`l=[10,22,-3.5,0]`
	`print('lista inicial:')`
	`print(l)`
	`print('lista ordenada descendente:')`
	`l=sorted(l,reverse=True)`
	`print(l)`
	
	Salida:
	`lista inicial:`
	`[10, 22, -3.5, 0]`
	`lista ordenada descendente:`
	`[22, 10, 0, -3.5]`
	`>>>`

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:

```
l.sort(reverse=True)
```
y

```
l.sort()
l.reverse()
```
producen el mismo resultado

En cambio:

```
l.reverse()
l.sort()
```

No!

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:

Programa	Salida
l=[1,2,3,4,5,6] t=('a','b','c') print('l=',l) print('t=',t) a=list(reversed(l)) print('l inversa como lista =',a) b=tuple(reversed(l)) print('l inversa como tupla =',b) a=list(reversed(t)) print('t inversa como lista =',a) b=tuple(reversed(t)) print('t inversa como tupla =',b) txt='hola' print(txt) palabra=list(txt) palabra=list(reversed(palabra)) txt=''.join(palabra) print('invertido=',txt)	 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 >>>

Usar Criterios de Ordenamiento no Estándar

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

Programa	Salida

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

¿Qué pasó acá? La lista no quedó ordenada alfabéticamente. Que mal!

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. El viejo dicho: *"Peras con peras"*.

Veamos cómo podemos realizar esto:

Programa	**Salida**

`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)`
	`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.

Ahora sí:

Programa	**Salida**

`lista=['Juan','ana','sergio','ELEna','ELEONORA','anALía']`
`print('Lista Inicial')`
`for n in lista:`
`  print (n)`
`lista.sort(key=str.lower)`
`print()`
 	
`print('Lista Ordenada')`
`for n in lista:`
`  print (n)`
	`Lista Inicial`
	`Juan`
	`ana`
	`sergio`
	`ELEna`
	`ELEONORA`
	`anALía`
	
	`Lista Ordenada`
	`ana`
	`anALía`
	`ELEna`

	ELEONORA Juan sergio >>>

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**

`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)`
	Lista Inicial
	Juan
	ana
	sergio
	ELEna
	ELEONORA
	anALía
	
	Lista Ordenada
	ana
	Juan
	ELEna
	sergio
	anALía
	ELEONORA
	>>>

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**() quienes 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**

`def cantVoc(pal):`
`  pal=pal.lower()`
`  cant=0`
`  for v in ('a','e','i','o','u','á','é','í','ó','ú'):`
`  cant+=pal.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=cantVoc`**`)`
`print()`
 	
`print('Lista Ordenada')`
`for n in lista:`
`  print (n)`
	`Lista Inicial`
	`Juan`
	`ana`
	`sergio`
	`ELEna`
	`ELEONORA`
	`anALía`
	
	`Lista Ordenada`
	`ELEONORA`
	`anALía`
	`sergio`
	`ELEna`
	`Juan`
	`ana`
	`>>>`

Otro Tip!
La función propia que definamos para criterio de ordenamiento (**cantVoc**, 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 (**cantVoc**, en este caso) el argumento cada vez que compare elementos. Y **sólo le enviará un elemento de la Estructura por vez.**

Algunas perlitas finales

Mirá estas dos funciones predefinidas adicionales: **map**() y **filter**()

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

Sintaxis de map:

map(función,secuencia**)**

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 (la función devolvió True)

Sintaxis de filter:

filter(función,secuencia)

Nota:

Tanto **map**() como **filter**() en Python 3.x devuelven una cosa rara (como **reversed**(), te acordás?). En realidad es algo así como un mapa de recorrido, con la lista de direcciones que se deben visitar ¿Cómo volvemos normal esto? Simple, aplicamos un casteo y convertimos la salida a lista. Y ya!

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 incluído en la función. Por eso se llaman funciones comprimidas.

Programa	**Salida**

`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)`
	`Lista Original`
	`[1, 2, 3, 4, 5, 6]`
	
	`Lista de Cuadrados`
	`[1, 4, 9, 16, 25, 36]`
	
	`Lista de Pares`
	`[2, 4, 6]`
	`>>>`

Bibliografía Adicional

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