



Departamento de Informática
Universidad Técnica Federico Santa María

Ayudantía

Programación de Computadores

IWI-131

Tema: Resumen Certamen 2

Fecha: 23/10/2017

Autor: Gonzalo Fernández - **@DevTotal**

1. Inicialización y Lectura de Listas

Las listas son un tipo de variable en Python que nos permite almacenar datos de manera ordenada (comenzando desde el **0**), sin importar el tipo de estos (pueden incluso ser otras listas):

Podemos inicializar una lista de 3 maneras:

<code>lista = []</code>	Inicializar lista vacía.
<code>lista = list()</code>	Inicializar lista vacía.
<code>lista = [1, 3, 4]</code>	Inicializar lista con valores.

El acceso a los elementos de una lista se realiza de la siguiente manera:

```
lista[inicio:final:salto]
```

- `inicio`: Posición del elemento desde donde se comenzará a leer la lista.
- `final`: Posición del elemento **posterior** al elemento donde se terminará de leer la lista.
- `salto`: Intervalo de elementos por los que se va a ir leyendo la lista, si es **negativo** entonces la lista será leída de derecha a izquierda.

Se pueden omitir valores de modo de conseguir partes específicas de la lista.

En los siguientes ejemplos se entenderán mejor estos conceptos habiendo definido `lista = [5, 3, 2, 1, 9, 3, 4]`:

```
lista[0]      # 5
lista[1]      # 3
lista[999]    # Error
lista[-1]     # 4, notar que es el último elemento
lista[-3]     # 9, notar la posición de este elemento.
```

```
# Recordar que lista = [5, 3, 2, 1, 9, 3, 4]

lista[1:1] # [], notar que entre el inicio (1) y
           el final (1), no hay elementos, pues,
           el final, es el elemento posterior al
           último elemento que se consultará, por
           ende, entre el inicio y el último
           elemento a consultar, no hay nada.

lista[1:1:1] # [], leer nota anterior, notar que si
             salto es 1, no altera la lectura de
             elementos de la lista.

lista[1:2] # [3], notar que 2 es el elemento
           posterior al elemento del inicio.

lista[-1:1] # [], notar que esto equivale a
            consultar entre el elemento -1 y el
            elemento 0 de la lista (0 porque 1 es
            el elemento posterior), pero como la
            lectura es de izquierda a derecha,
            entre el final y el inicio no hay nada.

lista[-1:7] # [4], notar que la lista tiene 7
            elementos, sabemos por el ejemplo
            anterior que en la posición -1 está el
            número 4, además, sabemos que 7 es la
            posición posterior al último elemento
            de la lista, que también es el 4, por
            ende, el resultado es [4].

lista[-3:7] # [9, 3, 4], notar que -3 equivale a la
            posición del número 9, luego, de allí,
            continúa recorriendo la lista hacia la
            derecha, donde estaría el último
            elemento.

lista[-7:7] # [5, 3, 2, 1, 9, 3, 4], notar que -7
            equivale a la posición del primer
            elemento, luego, de allí, continúa
            recorriendo la lista hacia la derecha
            hasta la posición antecesora al final,
            esto es, el último elemento!.
```

```
# Recordar que lista = [5, 3, 2, 1, 9, 3, 4]

lista[-6:-1] # [3, 2, 1, 9, 3], notar que el
              elemento -6 es 3 y el elemento -1 es 4
              (el último), entonces, como estamos
              recorriendo la lista de izquierda a
              derecha, obtenemos el resultado
              mostrado.

lista[0:7:-1] # [], notar que ahora el salto es
              negativo, por ende, estamos leyendo la
              lista de derecha a izquierda.

lista[6:0:-1] # [4, 3, 9, 1, 2, 3], notar que como
              leemos de derecha a izquierda, el
              resultado también está en ese orden
              (usé el 6 porque en el inicio se indica
              el elemento exacto desde el que se
              comenzará a leer, por ende, si quiero
              empezar del último, debo usar el 6 que
              es la última posición válida de la
              lista).

lista[6:0:-2] # [4, 9, 2], notar que como aumentó el
              salto, ahora la lectura es cada 2
              números.

lista[6:0:-9] # [4], notar que usamos un salto muy
              grande y por ende, no hay más elementos
              a excepción del primer elemento en el
              que comenzará la lectura.

lista[::2]    # [5, 2, 9, 4], como vimos
              anteriormente, podemos omitir valores
              al momento de leer nuestra lista.

lista[:2]    # [5, 3], omitimos el salto, por ende
              será 1, además, por defecto el inicio
              será 1, lo único que hemos declarado es
              el fin que es la posición del elemento
              posterior a la posición donde se
              terminará de leer la lista.
```

```
lista[0:]      # [5, 3, 2, 1, 9, 3, 4], omitimos el
               fin pero por defecto es el elemento
               posterior al último elemento de la
               lista, por ende, el resultado es toda
               la lista.
```

Resumiendo, sea e_n un elemento en la posición n de la lista:

- `lista[a]` devuelve el elemento a de la lista si $a > 0$, de lo contrario, será el elemento $M + a$ con M igual al total de elementos de la lista (para la lista que usamos de ejemplo, M sería 6).
- `lista[a:b]` devuelve una lista con los elementos $[e_a, \dots, e_{b-1}]$ de la lista.
- `lista[a:b:c]` asumiendo un k entero que aumenta de 1 en 1, devuelve los elementos $[e_a, e_{(a+c*k)}, \dots, e_{(a+c*k) < (b-1)}]$ de la lista, entendiendo que $k*c$ es la forma en que va saltándose números.

2. Operaciones sobre Listas

- Asignación: `lista[posición] = valor`
- Suma de listas: `lista_final = lista1 + lista2`
- Comparar si 2 listas son iguales: `lista1 == lista2`
- Eliminar un elemento de una lista: `del lista[a:b:c]`
- Ordenar una lista: `lista.sort()`
- Invertir una lista: `lista.reverse()`
- Insertar un elemento al final de una lista: `lista.append(valor)`
- Insertar un elemento en una posición dada: `lista.insert(valor, posición)`
- Obtener la posición de un elemento en la lista:
`lista.index(valor)`
- Sumar todos los números de una lista: `sum(lista)`
- Comprobar si un elemento está en una lista: `valor in lista`
- Clonación de listas: `lista_nueva = list(lista_a_clonar)`
- Obtener el largo de una lista: `len(lista)`

3. Iterar sobre Listas

Existen 2 maneras de iterar listas, ya sea con el uso del `for`, ó, el `while`:

- Iterar lista con **for**:

```
for i in lista:  
    print i
```

La variable **i** tendrá cada valor de la lista a medida que se itera sobre esta, el resultado del código anterior sería mostrar en pantalla cada valor contenido en la lista.

Ventajas de esta forma:

1. Es fácil de implementar.
2. Nos permite obtener instantáneamente el valor de la lista en una variable.
3. Podemos trabajar el valor de **i** sin modificar el valor en la lista.

Desventajas de esta forma:

1. No podemos detener la iteración a menos que usemos **return**.
2. No podemos saber la posición del elemento **i** en la lista a menos que usemos `lista.index(i)`.
3. No podemos avanzar o retroceder en la lista a voluntad.

- Iterar lista con **while**:

```
i = 0  
while i < len(lista):  
    print lista[i]  
    i += 1
```

La variable **i** tendrá cada posición de cada elemento de la lista, a medida que va aumentando hasta llegar al largo de la lista, notar que una lista de 5 elementos tiene un largo de 5 pero **i** llegaría a ser 4 ya que la lectura de datos en una lista parte del 0, entonces, **i** sería 0, 1, 2, 3, 4.

Ventajas de esta forma:

1. Tenemos control sobre la posición donde estamos iterando (aumentando o disminuyendo `i`).
2. Podemos detener el while haciendo que la condición de este ya no se cumpla (ejemplo básico: igualar `i` a infinito).

Desventajas de esta forma:

1. Cada vez que trabajemos un valor de la lista, debemos llamarlo con `lista[i]` lo cual, si bien no es grave, puede hacer que el código se vuelva ilegible.
2. Trabajar un valor implicaría modificar el valor en la lista, a menos que efectivamente sea ese nuestro objetivo, ó, guardemos el valor extraído de `lista[i]` en otra variable para así trabajar esa variable .
3. Es más difícil de usar, pues, según lo que hagamos dentro del while, podemos salirnos de la lista y causar un error, aunque claro, esto dependerá netamente del código ejecutado dentro del while, lo ideal es pensar bien lo que se está realizando.

Luego de esta comparación suele entrar la duda, ¿Es mejor usar el for sobre el while ?, ó, ¿Hay situaciones en que es mejor usar while que for? La respuesta a estas preguntas sólo se las dará la práctica, se pueden realizar exactamente las mismas cosas con ambos, ahora, el uso de uno por sobre el otro depende netamente de lo que le acomode más al programador, a modo personal, me gusta usar for cuando no necesito saber en qué posición de la lista estoy, en cambio, cuando necesito control sobre la posición de la lista donde estoy trabajando, hago uso del while, pero al final, se puede hacer exactamente lo mismo en ambos, ya sea en el for contando la posición en la que nos encontramos de la lista con una variable auxiliar, como en el while, asignando instantáneamente a una variable auxiliar el valor de `lista[i]`.

4. Tuplas

5. Operaciones sobre Tuplas

6. Iterar Tuplas

7. Conjuntos

8. Operaciones sobre Conjuntos

9. Iterar Conjuntos

10. Diccionarios

11. Operaciones sobre Diccionarios

12. Iterar Diccionarios
