Programación II
El modelo iterativo

Lic. Federico Severino Guimpel
LCC - FCEIA - UNR

Tópicos

- Listas
  - Definición
  - Operador [ ]
- Iteración
  - sobre listas
  - sobre range

Listas

La lista es una colección ordenada de elementos. Es equivalente a lo que en otros lenguajes se conoce por arrays, o vectores.

Las listas pueden contener cualquier tipo de dato: números, cadenas, booleanos, … y también listas.

Crear una lista es tan sencillo como indicar entre corchetes y, separados por comas, los valores que queremos incluir en la lista:

```
>>> a = ['pan', 'huevos', 100, 1234]
>>> a
['pan', 'huevos', 100, 1234]
```

Listas

Podemos acceder a cada uno de los elementos de la lista escribiendo el nombre de la lista e indicando el índice del elemento entre corchetes.

```
>>> a = ['pan', 'huevos', 100, 1234]

>>> a[0]
'pan'
>>> a[1]
'huevos'
>>> a[2]
100
>>> a[3]
1234
```

Listas

Una curiosidad sobre el operador [ ] de Python es que podemos utilizar también números negativos. Si se utiliza un número negativo como índice, esto se traduce en que el índice empieza a contar desde el final, hacia la izquierda; es decir, con [-1] accederíamos al último elemento de la lista, con [-2] al penúltimo, con [-3], al antepenúltimo, y así sucesivamente.

```
>>> a = ['pan', 'huevos', 100, 1234]
>>> a[-1]
1234
>>> a[-2]
100
>>> a[-3]
'huevos'
>>> a[-4]
'pan'
```

Otra cosa inusual es lo que en Python se conoce como *slicing* o particionado, y que consiste en ampliar este mecanismo para permitir seleccionar porciones de la lista. Si en lugar de un número escribimos dos números **inicio** y **fin** separados por dos puntos (inicio:fin) Python interpretará que queremos una lista que vaya desde la posición inicio a la posición fin, sin incluir este último.

```
>>> l
[99, True, 'una lista', [1, 2, 3, 4]]
>>> l[1:3]
[True, 'una lista']
>>> l[0:2]
[99, True]
```

Si escribimos tres números (inicio:fin:salto) en lugar de dos, el tercero se utiliza para determinar cada cuántas posiciones añadir un elemento a la lista.

```
>>> l = [99, True, "una lista", [1, 2,3,4]]
>>> l[0:4:2]
[99, 'una lista']
```

Hay que mencionar así mismo que no es necesario indicar el principio y el final del slicing, sino que, si estos se omiten, se usarán por defecto las posiciones de inicio y fin de la lista, respectivamente:

```
>>> l = [99, True, "una lista", [1, 2, 3, 4]]
>>> l[1:]
[True, 'una lista', [1, 2, 3, 4]]
>>> l[:2]
[99, True]
>>> l[:]
[99, True, 'una lista', [1, 2, 3, 4]]
>>> l[::2]
[99, 'una lista']
```

Listas

También podemos utilizar estos mecanismo para modificar la lista:

```
>>> l = [99, True, "una lista", [1, 2, 3, 4]]
>>> l[0:2] = ["nuevos", "valores"] #modifico valores
>>> l
['nuevos', 'valores', 'una lista', [1, 2, 3, 4]]
>>> l[0:2]= [] #elimino elementos
>>> l
['una lista', [1, 2, 3, 4]]
>>> l[1:1] = ["una cadena"] #inserto elementos
>>> l
['una lista', 'una cadena', [1, 2, 3, 4]]
```

Iteración sobre listas

Analicemos el siguiente problema:

> “Escribir una función que, dada una lista, imprima todos los valores que la forman”

Vamos a escribir una función recursiva como solución a esto. El código de la misma podría ser:

```
>>> def imprimeValoresLista(lista):
        if lista != []:
                print(lista[0])
                imprimeValoresLista(lista[1:])
```

Iteración sobre listas

Si analizamos el código vemos que hay un if para controlar si la lista es vacía o no. Esto es porque queremos acceder al primer elemento y, para esto debemos estar seguros que ese elemento exista. Si no se realiza esa verificación y, la función se escribe sin el if, es decir,

```
>>> def imprimeValoresLista(lista):
        print(lista[0])
        imprimeValoresLista(lista[1:])
```

al ejecutarla va a saltar un error.

Iteración sobre listas

```
>>> imprimeValoresLista([2,4,6])
2
4
6
Traceback (most recent call last):
  File "<pyshell#65>", line 1, in <module>
    imprimeValoresLista([2,4,6])
  File "<pyshell#64>", line 3, in imprimeValoresLista
    imprimeValoresLista(lista[1:])
  File "<pyshell#64>", line 3, in imprimeValoresLista
    imprimeValoresLista(lista[1:])
  File "<pyshell#64>", line 3, in imprimeValoresLista
    imprimeValoresLista(lista[1:])
  File "<pyshell#64>", line 2, in imprimeValoresLista
    print(lista[0])
IndexError: list index out of range
```

El cual nos dice que tratamos de acceder a una posición que no existe en nuestra lista.

Iteración sobre listas

Ahora vamos a dar otra versión de la función anterior, la cual NO es recursiva. Para poder hacer esto, usamos la sentencia **for**.

```
>>> def imprimeValoresLista(lista):
        for elemento in lista:
            print(elemento)
```

Como podemos ver, la función toma una lista y, para cada elemento que hay en ella se ejecuta la sentencia print. Es decir, se realizan tantos print como elementos haya en la lista.

Iteración sobre listas

La palabra clave **`for`**, en el ejemplo anterior, representa una forma de recorrer una lista, usando una variable para esto.
La variable va tomando los diferentes valores que forman la lista, en orden, según la posición.

Esta forma de recorrer la lista se dice que es Iterativa y, cada vez que la variable pasa a un nuevo valor de la lista es una Iteración.

Veamos diferentes ejemplos de su uso y, analicemos cómo podemos usar esto para reescribir algunas funciones recursivas de forma Iterativa.

Iteración sobre listas

```
>>> def imprimeValoresLista(lista):
        for elemento in lista:
                print(elemento)
```

```
>>> imprimeValoresLista([2,4,6])
2
4
6
>>> imprimeValoresLista([])
>>> l = [99, True, "una lista", [1, 2, 3, 4]]
>>> imprimeValoresLista(l)
99
True
una lista
[1, 2, 3, 4]
```

Iteración sobre listas

Las listas permiten lo que se conoce como definición por comprensión, es decir, se crea una nueva lista aplicando una expresión a los elementos de otra lista que cumplan una determinada condición. La sintaxis de esta definición es:

```
[expresión for variables in iterable if condición]
```

```
>>> l = [2,4,6]
>>> [3 * x for x in l]
[6, 12, 18]
>>> [3 * x for x in l if x > 3]
[12, 18]
>>> [3 * x for x in l if x < 1]
[]
```

Iteración sobre rangos

Analicemos este ejercicio:

1. Calcular el factorial de un número

Este problema ya lo hicimos anteriormente usando funciones recursivas, en este caso, vamos a resolverlo usando **for**.
Para esto, vamos a hacer uso de una estructura llamada **range** que genera una lista de números sobre la cual se puede iterar usando **for**. Veamos un par ejemplos de esto.

Iteración sobre rangos

Range genera un rango de números enteros. Si se le pasa un solo argumento, se genera un rango desde el 0 hasta el argumento - 1.

```
>>> for i in range(3):
        print(i)


0
1
2
```

Iteración sobre rangos

Cuando se le pasan dos argumentos, el primero indica el inicio y, el segundo el fin (sin incluirlo).

```
>>> for i in range(1,5):
        print(i)


1
2
3
4
```

Iteración sobre rangos

Si pasamos tres parámetros, el tercero indica el step, la diferencia entre los números que se vayan generando en la secuencia.

```
>>> for i in range(1,5,2):
        print(i)


1
3
```

Iteración sobre rangos

Usando esto, la solución para escribir una función que reciba un número y muestre su factorial usando for sería:

```
>>> def factorialIterativo(n):
        factorial = 1
        for numero in range(1,n+1):
                factorial = factorial * numero
        print(factorial)


>>> factorialIterativo(5)
120
```

Iteración sobre rangos

Como vemos, usamos una variable para almacenar el resultado de la multiplicación.
Al finalizar el for va a tener el valor del factorial.
En cada iteración multiplicamos el valor que tiene la variable hasta el momento por el número que estamos iterando en la lista.
Finalmente, mostramos el valor de la variable.

¿Qué sucede si se llama a la función con un número negativo? ¿Cuál sería su salida?

Iteración sobre rangos

Tenemos ahora este ejercicio:

2. Imprimir los primeros 25 números naturales pares

Este problema, que es el ejercicio 1 de la práctica 1 lo podemos resolver usando for y range:

```
>>> def primeros25NumerosPares():
        for i in range(0,50,2):
            print(i)
```

Iteración sobre rangos

Cuya salida al ejecutarla es:

```
>>> primeros25NumerosPares()
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
32
34
36
38
40
42
44
46
48
```