# Recorridos secuenciales de todos cumplen o alguno cumple. Operaciones de conjuntos. Funciones zip y enumerate. Funciones anónimas.

Fundamentos de Programación
Departamento de Lenguajes y Sistemas Informáticos

2024/2025

# Recorrido secuencial de “todos cumplen”

Esquema de recorrido que permite ver *si todos* los elementos de un contenedor cumplen una *condición*:, aprovechando que *break* interrumpe (termina un bloque for o un bloque while), hacemos más eficiente el algoritmo:

```
def todos_cumplen_que...(contenedor):
    res=True
    for elemento in contenedor:
        if not condición:
            res=False
            break
    return res
```

Presuponemos que todos cumplen

Sentencia que rompe la ejecución del bucle

Se presupone que todos cumplen la condición (res=*True*). Se van recorriendo todos los elementos y, si alguno no cumple la condición (ya todos no cumplen), por lo que se cambia el valor de res (res=*False*) y no se sigue preguntando (*break*), lo que hace más eficiente el algoritmo.

# Recorrido secuancial de “existe alguno”:

Esquema de recorrido que permite ver *alguno* de los elementos de un contenedor cumplen una *condición*:, aprovechando que *break* interrumpe (termina un bloque for o un bloque while), hacemos más eficiente el algoritmo:

```
def hay_alguno_que…(contenedor):
    res=False
    for elemento in contenedor:
        if condición:
            res=True
            break
    return res
```

Se presupone que ninguno cumple

Sentencia que rompe la ejecución del bucle

Se presupone que ninguno cumple la condición (res=*False*). Se van recorriendo todos los elementos y si alguno cumple la condición ya se ha encontrado uno que la cumple, por lo que se cambia res (res=*True*) y no se sigue preguntando (*break*) lo que hace más eficiente el algoritmo

# Ejercicio para realizar con notebooks

Dada las listas de números *mi_lista1=[4, 2, 20, 12, 8, -10]* y *mi_lista2=[4, 2, 17, 11, 8, -10]*

Realizar y probar las funciones que respondan a las siguientes preguntas para ambas listas:

- ¿son_todos_pares?
- ¿alguno_múltiplo_de_3?

# Operadores para el manejo de conjuntos

Si se dispone de dos conjuntos:

- conjunto1={2,4,4,7.2}
- conjunto2={8.2,4,2,'a','2'}

*Operador unión* (|) *(equivalente a 'or')*: Los elementos de ambos sin repetir

```
conjunto1|conjunto2 → {2, 4, 7.2, 8.2, 'a', '2'}
```

*Operador intersección* (&) *(equivalente a 'and')*: Los elementos comunes sin repetir

```
conjunto1&conjunto2 → {2,4}
```

*Operador diferencia* (-) Los elementos del primero que no están en el segundo

```
conjunto1-conjunto2 → {7.2}
conjunto2-conjunto1 → {8.2,'2','a'}
```

*Operador diferencia simétrica* (^) Los elementos que solo están en uno de los conjuntos=U-∩

```
conjunto1^conjunto2 → {'a',7.2,8.2,'2'}
conjunto2^conjunto1 → {'a',7.2,8.2,'2'}
```

# Operadores para el manejo de conjuntos

Si se dispone de tres conjuntos:

- conjunto1={2,4,4,7.2}
- conjunto2={8.2,4,2,'a','2',7.2}
- conjunto3={'a','a',8.2,4,2,'2',7.2}

## *Operador de igualdad* (==)

conjunto1==conjunto2 → False

conjunto2==conjunto3 → True *(no entran repetidos y no importa el orden)*

## *Operador subconjunto* (< o <=)

conjunto1<conjunto2 → True

conjunto2<conjunto1 → False

conjunto2<conjunto3 → False

conjunto2<=conjunto3 → True

# Otras funciones para recorrer contenedores: ZIP

En Python, además de la función range(*inicio*, *final*, *paso*) que permite generar una secuencia de valores desde el *inicio* (incluido), hasta el *final* (sin incluir), con un incremento de *paso*.
*Existen otras dos funciones importantes*:

- *zip* (contenedor 1, contenedor 2, contenedor 3, …):
  Devuelve, en cada iteración *una tupla* con un elemento de cada contenedor, mientras haya elementos de todos los contenedores. En pocas palabras: "*va recorriendo simultáneamente un elemento de cada contenedor*". Se suele usar para analizar/comparar los valores de contenedores de forma "sincronizada"

  *Ejemplo*:

```
for tupla in zip ([1,2,3],"Joselito",['a','b','c','d']):
    print (tupla)
```

→

```
(1,'J','a')
(2,'o','b')
(3,'s','c')
```

¡*Termina el **for** cuando se haya recorrido el contenedor con menos elementos*!

# Otras funciones para recorrer contenedores: ZIP

*Ejercicio típico:*

Dado un contenedor obtener datos de relación entre cada dos elementos consecutivos del contenedor.

Sea la lista *mi_lista=[12, 30, 11, -5, -9, 44]* devolver la relación entre dos elementos consecutivos en el orden natural

```
mi_lista=[12, 30, 11, -5, -9, 44]
res=list()
lista_ordenada=sorted(mi_lista)
for e1, e2 in zip(lista_ordenada[1:],lista_ordenada[0:]):
  res.append(e2/e1)
print(lista_ordenada)
print(res)
```

```
[-9, -5, 11, 12, 30, 44]
[1.8, -0.4545454545454545, 0.9166666666666666, 0.4, 0.6818181818181818]
```

# Otras funciones para recorrer contenedores: Enumerate

- *enumerate* (contenedor):

  Devuelve, en cada iteración una tupla con dos elementos (*añade un nuevo primer elemento*):
  - El 1º es un contador o posición, que empieza en 0 (se genera automáticamente)
  - El 2º es el elemento correspondiente del contenedor.

*Ejemplos:*

```
for elemento in "ejemplo":
    print (elemento)
```

```
e
j
e
m
p
l
o
```

```
for tupla in enumerate ("ejemplo"):
    print(tupla) o si ponemos print(tupla[0],tupla[1])
```

```
(0, 'e')        0 e
(1, 'j')        1 j
(2, 'e')        2 e
(3, 'm')        3 m
(4, 'p')        4 p
(5, 'l')        5 l
(6, 'o')        6 o
```

```
for i, elemento in enumerate ("ejemplo"):
    print (i, elemento)
```

```
0 e
1 j
2 e
3 m
4 p
5 l
6 o
```

# Combinando enumarate y zip

*Ejemplos que combina enumerate y zip:*

```
for posición, tupla in enumerate (zip([1,2,3],"Joselito",['a','b','c','d’])):
    print ("La posición de",tupla,"es la",posición)
```

```
La posición de (1, 'J', 'a') es 0
La posición de (2, 'o', 'b') es 1
La posición de (3, 's', 'c') es 2
```

*Observar que el for termina cuando se acaba uno de los contenedores*

```
for tupla in enumerate (zip([1,2,3],"Joselito",['a','b','c','d’])):
    print ("La posición de", tupla[1],"es la", tupla[0])
```

```
La posición de (1, 'J', 'a') es 0
La posición de (2, 'o', 'b') es 1
La posición de (3, 's’, 'c') es 2
```

# Funciones denominadas : sin nombre, anónimas o lambdas

Python permite *pasar como parámetro una función*. Esa función puede estar predefinida como hemos visto hasta ahora *o construirla en el momento de pasarla como parámetro*. En este último caso, se denominan funciones *sin nombre, anónimas o lambda*.

Supongamos el siguiente código:

```
from typing import List
from math import pow
def modifica_lista(datos:List)->List:
  res=list()
  for elemento in datos:
    res.append(elemento*2+pow(elemento-7.5,3))
  return res
```

*Tranformación*
Se multiplica por 2 cada elemento y se le suma la potencia de 3 de dicho elemento menos 7.5)

```
mis_números=[1,3,5,5,7]
print (modifica_lista(mis_números))
```

Que proporciona la siguiente salida

```
[-272.625, -85.125, -5.625, -5.625, 13.875]
```

# Funciones denominadas : sin nombre, anónima o lambda

Vemos que la función *modifica_lista* solo realiza transformación descrita en la dispositiva anterior. Ahora bien, le podemos dar *mayor versatilidad si le añadimos* un segundo parámetro que sea *una función sin nombre, anónimas o lambda* para que realice otras transformaciones.

```
from typing import List
from math import pow
def modifica_lista(datos:List, mi_función)->List:
  res=list()
  for elemento in datos:
    res.append(mi_función(elemento))
  return res
```

```
mis_números=[1,3,5,5,7]
print (modifica_lista(mis_números, lambda x:x*2+pow(x-7.5,3)))
print (modifica_lista(mis_números, lambda e:e*2 ))
```

Que proporciona las siguientes salidas

```
[-272.625, -85.125, -5.625, -5.625, 13.875]
[2, 6, 10, 10, 14]
```