Herramientas de programación en Python

2. Tipos de datos estructurados

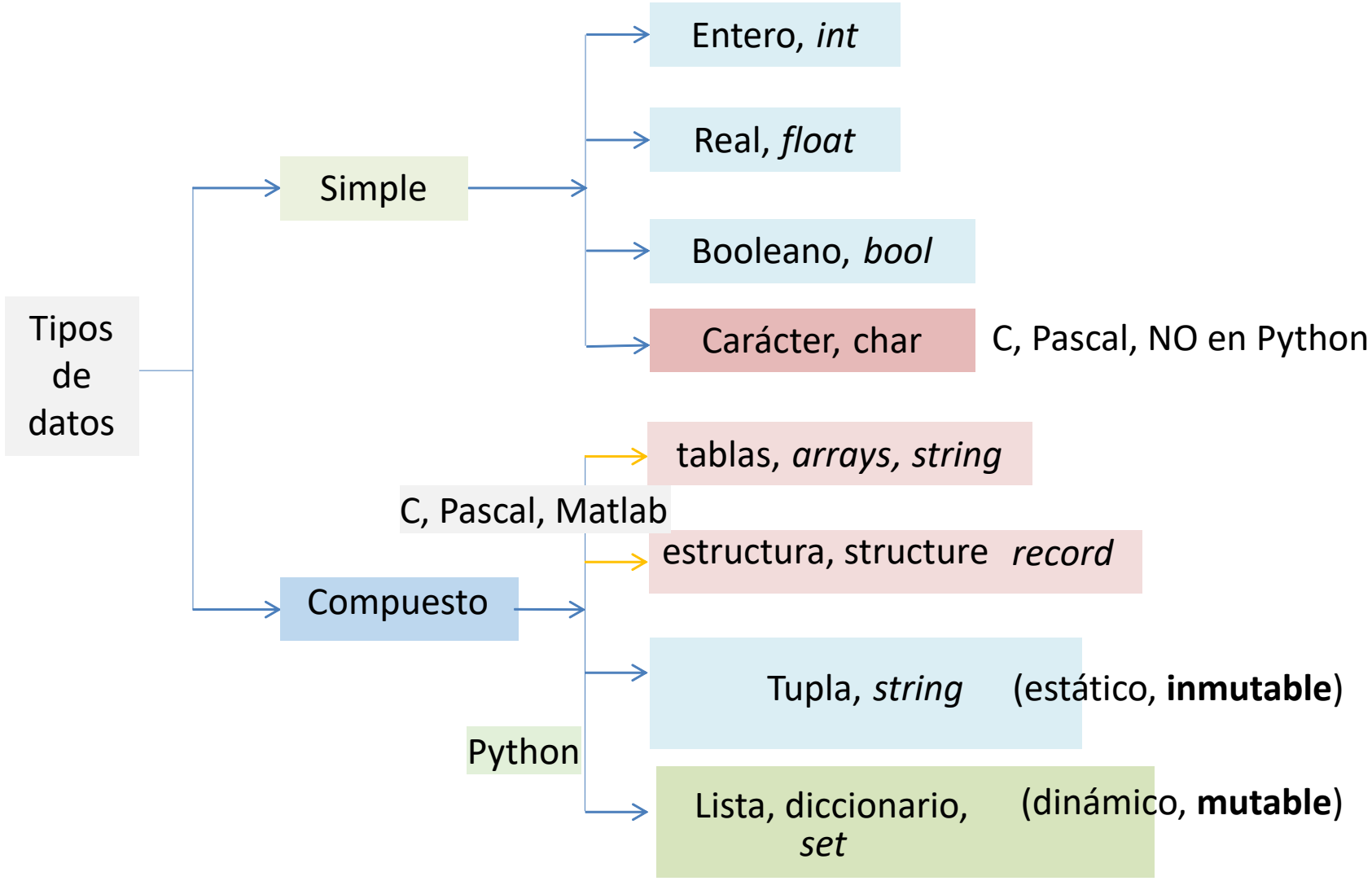
2.2 homogéneos y heterogéneos, dinámicos: listas

- Operadores sobre listas
- Métodos
- *List Comprehension*
- Iterador zip
- Listas anidadas (listas de listas) homogéneas y heterogéneas

Pedro Gomis

pedro.gomis@upc.edu

Tipos de datos



Tipos de datos

for elemento in secuencia:
    Instrucciones

Secuencias típicas en Python:

- **range(n)** , ejemplo: **range(5)** → secuencia: 0,1,2,3,4
- **String,** ejemplo 'Hola' → secuencia 'H','o','l','a'
- **Tupla**, ejemplo (3, 5, True) → secuencia 3, 5, True
- **Listas**, ejemplo [4, 7, 'Hola'] → secuencia 4, 7, 'Hola'

Datos estructurados mutables (dinámicos)

- A diferencia de los ***string***, **tuplas** y *conjuntos congelados*, los datos estructurados mutables (dinámicos) se caracterizan porque sus elementos **pueden cambiar de valor y se puede añadir o eliminar elementos**.
- Datos estructurados mutables (en Python): las **listas**, los conjuntos (**Set**) y los **diccionarios**.

Listas (*mutables*)  **Tuplas** (*inmutables*)

Set (*mutables*)  **Frozenset** (*mutables*)

Diccionarios

Listas

- Las **listas**, así como las **tuplas** y ***strings***, están formadas por una **secuencia de datos**. Pero sus elementos pueden ser modificados, eliminarse o aumentarse. SON MUTABLES
- Los elementos de las **listas** pueden ser datos simples (numéricos o booleanos), ***strings***, **tuplas** u otras **listas**.
- Los elementos se indexan y rebanan igual que los ***strings*** y **tuplas**, a través de un número entero.
- La sintaxis de las **listas** es una secuencia de valores separados por comas encerrados entre corchetes. Ejemplos:

```
>>> v1 = [2, 4, 6, 8, 10]
>>> type(v1)
<class 'list'>
>>> v2 = [7, 8.5, 'a', 'Hola', (2, 3), [11, 12]]
>>> v2
[7, 8, 'a', 'Hola', (2, 3), [11, 12]]
>>> juegos = ['tennis','baseball','football','voleyball','natación']
>>> juegos
['tennis', 'baseball', 'football', 'voleyball', 'natación']
```

Listas

- Podemos generar una lista con una secuencia de números enteros con el tipo de datos **range**(), de la forma,

```
>>> v = list(range(1,11))
>>> v
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

- La función interna **`list`**() también sirve para convertir tipos de datos iterables, como ***strings*** o **tuplas** a tipo **lista**. Se puede crear también una lista vacía. Ejemplos:

```
>>> t = (1, 2, 3)          # t es una tupla
>>> list(t)
[1, 2, 3]                  # convertida a lista
>>> s = 'Hola'
>>> list(s)
['H', 'o', 'l', 'a']
>>> e = list()             # lista vacía
>>> e = []                 # lista vacía
```

Indexación, recorte y otras operaciones de listas

- En las listas, el acceso a sus elementos, la extracción de elementos y las operaciones se realizan de la misma forma que en los ***strings*** y las **tuplas**. Los operadores de corte (***slice***) `[n:m]` se usan también en las **listas**:

```
>>> v2 = [7, 8, 'a', 'Hola', (2,3), [11, 12]]
>>> v2[0]
7
>>> v2[-1]
[11, 12]
>>> v2[-2]
(2, 3)
>>> v2[0:3]
[7, 8, 'a']
>>> t = ['las','listas','son','mutables']
>>> t[3] = 'dinámicas'
>>> t
['las', 'listas', 'son', 'dinámicas']
>>> len(t)
4
```

Se puede observar la mutabilidad de las listas

Operadores +, *, in, not in

- Las **listas** pueden concatenarse y repetirse con los operadores + y *, respectivamente, como los ***string*** y **tuplas**,

```
>>> v1 = [2, 4, 6, 8, 10]
>>> v3 = [3, 5, 7]
>>> v1 + v3
[2, 4, 6, 8, 10, 3, 5, 7]
>>> 3*v3
[3, 5, 7, 3, 5, 7, 3, 5, 7]
```

- La composición iterativa **for** - **in** de Python también se usan con **listas**, ya que éstas son una secuencia de elementos.

- Los operadores booleanos **in** y **not in** evalúan si un elemento pertenece o no a una secuencia (***string***, **tupla** o **lista**). Ejemplos con **listas**:

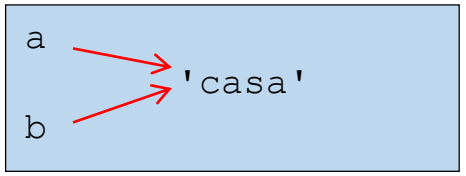
```
>>> v2 = [7, 8, 'a', 'Hola', (2,3), [11, 12]]
>>> 8 in v2
True
>>> 'Hola' in v2
True
>>> 27 in v2
False
>>> 'HOLA' not in v2
True
```

Operador is. Objetos, valores y referencias

- Se dispone en ***Python*** del operador **is** que indica si dos variables están **referidas** al mismo **objeto** o no. Si ejecutamos las instrucciones:

```
>>> a = 'casa'
>>> b = 'casa'
>>> id(a)
123917904
>>> id(b)
123917904
>>> a is b
True
```

podemos ver que ambas variables `a` y `b` están **referidas** al mismo **objeto**, que tiene valor `'casa'` y ocupa la posición de memoria `123917904` (esta posición es arbitraria). La instrucción `a is b` es cierta.

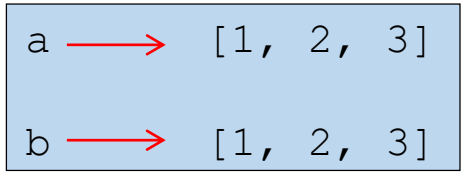


En los tipos de dato ***string*** (inmutables), Python crea solo **un objeto** por economía de memoria y ambas variables están referidas al mismo objeto.

Objetos, valores y referencias

- Aunque se formen **dos listas** con los mismos valores, Python crea **dos objetos**, que ocupan diferente posición de memoria:

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> id(a)
123921992
>>> id(b)
123923656
>>> a is b
False
```



- Las listas asignadas a las variables a y b, aunque con el mismo valor, son **objetos diferentes**.

Objetos, valores y referencias

- En el siguiente ejemplo al asignar una variable (con una lista) a otra, no se crea otro objeto, sino que el copiado se refiere al mismo objeto:

```
>>> a = [1, 2, 3]
>>> b = a
>>> id(b)              # a y b --> [1, 2, 3]
123921992
>>> a is b
True
```

- La variable `b` es un ***alias*** de `a` y están **referenciadas**. Si modificamos o añadimos un valor al objeto [1, 2, 3], a través de una de las variables, entonces modificamos la otra:

```
>>> b[0] = 15
>>> a
[15, 2, 3]
```

- Si queremos **copiar** una variable de otra (y que sean **objetos diferentes**), se disponen del método **copy**, que se presentará a continuación, en métodos de listas. También se pudiera hacer:

```
>>> b = a[:]
```

y la variable **b** contiene la misma lista que **a**, pero es un objeto diferente.

Métodos de las listas

- Las **listas** (mutables) disponen de un amplio grupo de **métodos** asociados que permiten añadir nuevos elementos, quitarle elementos, ordenarlos, etc. Se usa el operador punto (**.**) para acceder a los métodos de los objetos

```
>>> v = [1, 2, 3, 4, 5]
>>> v.append(6)                  # añade un elemento al final de la lista, en este caso el 6
>>> v
[1, 2, 3, 4, 5, 6]
>>> v2 = v.copy()                # copia la lista en otro objeto
>>> v.count(3)                   # cuenta las veces que aparece el elemento (3)
1
>>> v.extend([7, 8, 9])          # extiende la lista con los elementos de otra
>>> v
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> v.index(5)                   # índice de la primera ocurrencia del valor (5)
4
>>> v.insert(2, 4)               # insert(índice, valor), inserta 4 en posición 2
>>> v
[1, 2, 4, 3, 4, 5, 6, 7, 8, 9]
>>> v.pop(0)                     # Remueve y devuelve elemento de posición (0)
1                                # si se usa v.pop(-1) o v.pop() remueve el último
```

Métodos de las listas

```
>>> v
[2, 4, 3, 4, 5, 6, 7, 8, 9]
>>> v.remove(4)                  # remueve la 1ra ocurrencia del valor (4)
>>> v
[2, 3, 4, 5, 6, 7, 8, 9]
>>> v.reverse()                  # invierte el orden de los elementos de la lista
>>> v
[9, 8, 7, 6, 5, 4, 3, 2]
>>> v1 = [4, -3, 5, 0, 1]
>>> v1.sort()                    # ordena en orden ascendente
>>> v1
[-3, 0, 1, 4, 5]
>>> v1 = [4, -3, 5, 0, 1]
>>> v1.sort(reverse=True)        # ordena en orden descendiente
>>> v1
[5, 4, 1, 0, -3]
>>> v.clear()                    # remueve todos los elementos de la lista
>>> v
[]
```

Métodos de string – listas: split - join

```
>>> str = '10/04/2018'
>>> lst = str.split('/')
>>> lst
['10', '04', '2018']
>>> str2 = '/'.join(lst)
>>> str2
'10/04/2018’
>>> help(str.split)
Help on built-in function split:
split(...) method of builtins.str instance
    S.split(sep=None, maxsplit=-1) -> list of strings

    Return a list of the words in S, using sep as the
    delimiter string.  If maxsplit is given, at most maxsplit
    splits are done. If sep is not specified or is None, any
    whitespace string is a separator and empty strings are
    removed from the result.
>>> S = '/'
>>> help(S.join)
Help on built-in function join:

join(...) method of builtins.str instance
    S.join(iterable) -> str

    Return a string which is the concatenation of the strings in the
    iterable.  The separator between elements is S.
```

Listas - Ejemplos

1) Producto escalar de 2 vectores*

```
def ProdEsc(v1,v2):
    """ Funcion que calcula el producto escalar de 2 vectores
    >>> ProdEsc([1,1,1],[2,2,2])
    6
    """
    n1 = len(v1)
    suma = 0
    for i in range(n1):
        suma += v1[i]*v2[i]
    return suma

print(ProdEsc([1, 2],[3, 4]))
print(ProdEsc([1,1],[4,5]))
```

$$suma = \sum_{i=0}^{n1-1} v1[i] * v2[i]$$

```
>>>
11
9
```

* En el siguiente apartado, **array** de NumPy, veremos que los objetos **array** son mucho más adecuados que las lista para operar con vectores

Iterador zip (del inglés zipper –cremallera)

- Función zip: Es un iterador que opera sobre varios iterables, y crea tuplas agregando elementos de las secuencias iterables (por ejemplo, strings).

- Aplicación de zip para recorrer 2 secuencias en paralelo.

```
def CuentaElemsMismaPosición(s1,s2):
    """ Dice cuántas letras iguales están en la misma posición en 2
        palabras s1 y s2. Se pueden usar listas o tuplas
    >>> CuentaElemsMismaPosición('Hola', 'Colado')
    3
    """
    contador = 0
    for c1, c2, in zip(s1,s2):
        if c1 == c2:
            contador += 1
    return contador
```

```
>>> CuentaElemsMismaPosición('Hola', 'Adios')
0
>>> CuentaElemsMismaPosición('Hola', 'Colado')
3
```

Listas - Ejemplos

1) Producto escalar de 2 vectores (opción usando **zip**)

```
def ProdEsc(v1,v2):
    """ Funcion que calcula el producto escalar de 2 vectores
    >>> ProdEsc([1,1,1],[2,2,2])
    6
    """
    suma = 0
    for a,b in zip(v1,v2):
        suma += a*b
    return suma

print(ProdEsc([1, 2],[3, 4]))
print(ProdEsc([1,1],[4,5]))
```

$$suma = \sum_{i=0}^{n1-1} v1[i] * v2[i]$$

```
>>>
11
9
```

2.1) Suma de 2 vectores (opción 1: pre-asignar vector lista)

```
def SumaVect(v1,v2):
    """ Funcion que calcula la suma de 2 vectores
    >>> SumaVect([1,1,1],[2,2,2])
    [3, 3, 3]
    >>> SumaVect([1,1],[2,2,2])
    -1
    """
    n1 = len(v1)
    n2 = len(v2)
    if n1 != n2:                # chequea que sean del mismos tamaño
        vsuma = -1              # -1 indica: error, vectores distinto tamaño
    else:
        vsuma = n1*[0]          # vector de 0's, preasignado
        for i in range(n1):
            vsuma[i] = v1[i] + v2[i]
    return vsuma


print(SumaVect([1, 2],[3, 4]))
print(SumaVect([1,1],[4,5]))
```

```
>>>
[4,6]
[5,6]
```

2.1) Suma de 2 vectores (opción 2: añadiendo elementos a la lista)

```
def SumaVect(v1,v2):
    """ Funcion que calcula la suma de 2 vectores
    >>> SumaVect([1,1,1],[2,2,2])
    [3, 3, 3]
    >>> SumaVect([1,1],[2,2,2])
    -1
    """
    n1 = len(v1)
    n2 = len(v2)
    if n1 != n2:                # chequea que sean del mismos tamaño
        vsuma = -1              # -1 indica: error, vectores distinto tamaño
    else:
        vsuma = []              # vector (lista) vacío
        for i in range(n1):
            vsuma.append(v1[i] + v2[i])   # se añaden elementos sumados
    return vsuma


print(SumaVect([1, 2],[3, 4]))
print(SumaVect([1,1],[4,5]))
```

```
>>>
[4,6]
[5,6]
```

3) Función que dice si una secuencia es creciente (True) o no (False)

```
def es_creciente(v):
    """
    >>> es_creciente([])
    True
    >>> es_creciente([1])
    True
    >>> es_creciente([1,1,2,3,6])
    True
    >>> es_creciente([1,1,4,3,5,6])
    False
     """
    for i in range(len(v)-1):
        if v[i + 1] < v[i]:
            return False
    return True

print(es_creciente([1,2,3,5]))
print(es_creciente([1,3,2,4]))
```

```
>>>
True
False
```

4) Función que devuelve el primer valor negativo de una lista o 0 si no hay

```
def primer_negativo(nums):
    ''' Devuelve el primer negativo de la lista o 0 si no hay negativos
    >>> primer_negativo([3,5,1,-3,0,-7,8])
    -3
    >>> primer_negativo([3,5,1,1,0,7,8])
    0
    '''
    for n in nums:
        if n < 0:
            return n
    return 0

print(primer_negativo([3,5,1,-1,0,-7,8]))
print(primer_negativo([3,5,1,1,0,7,8]))
```

```
>>>
-1
0
```

5) Cuántas veces se tiene Temp ≥ 38 ºC (fiebre) de las medidas hechas. Consideramos que se tiene una medida por hora. Temp[0] es la temperatura en la hora 0, Temp[1] en la hora 1, ...

```
def VecesFiebre(v):
    """
    Función indica las veces que tiene fiebre de las medidas hechas
    >>> Temp = [37.2,38.1,39,36.8]
    >>> VecesFiebre(Temp)
    2
    >>> VecesFiebre([37.2,37.1,37,36.8])
    0
    """
    veces = 0
    for i in range(len(v)):
        if v[i] >= 38:
            veces += 1
    return veces

print(VecesFiebre([36,37,37.2,36.8,38,37.5,37.6,38.1,37,37.3]))
print(VecesFiebre([36,37,37.2,36.8,37,37.5,37.6,37.1,37,37.3]))
import doctest
print(doctest.testmod())
```

```
>>>
2
0
```

6) Búsqueda de la 1ª hora en que Temp ≥ 38 ºC (fiebre) de las horas medidas (0,1,2,..) .
Si no ha tenido fiebre devolver -1 (búsqueda)

```
def Hora1Fiebre(v):
    """
    Primera hora en que ha habido fiebre de las horas medidas (Si no -1)
    >>> Hora1Fiebre([37.2,37.3,38.1,39,36.8])
    2
    >>> Hora1Fiebre([37.2,37.1,37,36.8])
    -1
    """
    for i in range(len(v)):
        if v[i] >= 38:
            return i
    return -1

print(Hora1Fiebre([36,37,37.2,36.8,38,37.5,37.6,38.1,37,37.3]))
print(Hora1Fiebre([36,37,37.2,36.8,37,37.5,37.6,37.1,37,37.3]))
import doctest
print(doctest.testmod())
```

```
>>>
4
-1
```

6) Búsqueda de la 1ª hora en que Temp ≥ 38 ºC (fiebre) de las horas medidas (0,1,2,..).
Si no ha tenido fiebre devolver -1 (búsqueda) (Usando while para la búsqueda)

```
def Hora1Fiebre(v):
    """
    Primera hora en que ha habido fiebre de las horas medidas (Si no -1)
    >>> Hora1Fiebre([37.2,37.3,38.1,39,36.8])
    2
    >>> Hora1Fiebre([37.2,37.1,37,36.8])
    -1
    """
    hora = -1
    i = 0
    HayFiebre = False
    while i < len(v) and not HayFiebre:
        if v[i] >= 38:
            hora = i
            HayFiebre = True
        i += 1
    return hora

print(Hora1Fiebre([36,37,37.2,36.8,38,37.5,37.6,38.1,37,37.3]))
print(Hora1Fiebre([36,37,37.2,36.8,37,37.5,37.6,37.1,37,37.3]))
```

```
>>>
4
-1
```

Comprensiones de listas (list comprehension)

- ***list comprehension***: Es una herramienta para transformar una lista (o cualquier iterable) en otra lista.
- Tratan de emular la notación de ***conjuntos*** usada en matemáticas. Por ejemplo: {x | x > 10} o $\{ x^2 \mid x \in \mathbb{N} \}$
- La notación en Python es:

```
[f(x) for x in S if P(x)]
```

- ***Produce una lista que contiene los valores de la secuencia*** `S` ***seleccionada por el predicado*** `P` ***y mapeada por la función*** `f`. ***La sentencia*** `if` ***es opcional***, y pueden existir varias sentencias `for`, cada una con su propia cláusula opcional `if`, para representar bucles anidados.
- Ejemplo de crear lista de los cuadrados de la secuencia S = (1, 2, 3, 4, 5):

```
>>> [x**2 for x in (1, 2, 3, 4, 5)]
[1, 4, 9, 16, 25]
```

Comprensiones de listas (list comprehension)

- Ejemplo: Crear una lista de números quintuplicados del 1 al n:

```
Mult5 = []                              # Método clásico
for x in range(1,n+1):
    Mult5.append(5*x)
```

```
Mult5 = [5*x for x in range(1,n+1)]  # List comprehension
```

```
>>> n = 10
>>> [5*x for x in range(1,n+1)]
[5, 10, 15, 20, 25, 30, 35, 40, 45, 50]
```

Comprensiones de listas (list comprehension)

- Ejemplo: Filtrar (quitarle) a una lista sus números negativos:

```
lst = [1,2,-3,5,-6,0,12,-1,7]
lst2 = []                                   # Método clásico
for x in lst:
    if x>=0:
        lst2.append(x)
```

```
lst2 = [x for x in lst if x>=0]      # List comprehension
```

```
>>> lst = [1,2,-3,5,-6,0,12,-1,7]
>>> lst2 = [x for x in lst if x>=0]
>>> lst2
[1, 2, 5, 0, 12, 7]
```

Comprensiones de listas (list comprehension)

- Ejemplo: Listar primos entre 2 números:

```
def es_primo(n):
    if n <= 1: return False
    for d in range(2, n//2+1):
        if n % d == 0:
            return False
    return True
```

```
def list_primos(a,b):
    """
    >>> list_primos(2,23)
    [2, 3, 5, 7, 11, 13, 17, 19, 23]          # Método clásico
    """
    lst=[]
    for i in range(a,b+1):
        if es_primo(i):
            lst.append(i)
    return lst
```

```
def list_primosLC(a,b):                    # List comprehension
    return [x for x in range(a,b+1) if es_primo(x)]
```

Listas anidadas (o listas de listas)

- Una lista anidada es una lista donde sus elementos son a su vez listas.
- Son útiles para representar tablas o matrices de datos (datos homogéneos)* La matriz:

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

- Se puede representar en Python por:

```
>>> m = [[1, 2, 3, 4], [5, 6, 7, 8],[9, 10, 11, 12]]
```

- En un editor de código de Python se puede escribir saltando a la siguiente línea después de la coma que separa cada elemento de la lista (que representa una fila de la matriz):

```
m = [[1, 2, 3, 4],
     [5, 6, 7, 8],
     [9, 10, 11, 12]]
```

* En el siguiente apartado, **array** de NumPy, veremos que los objetos **array** son mucho más adecuados que las lista para operar con vectores y matrices

Listas anidadas (o listas de listas) para representar matrices

- El **primer elemento** de la lista es la **primera fila** de la matriz. El valor 7 es el elemento m_{23}, en Python es la posición `[1][2]`-> `m[1][2]`:

```
>>> m[0]        # fila 1
[1, 2, 3, 4]
>>> m[1][2]     # elemento m_23 de la matriz (en Python m_12)
7
```

$$\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}$$

- Probemos **2** matrices (con listas) e intentemos **sumarlas**; por ejemplo,

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix}$$

```
>>> m1 = [[1, 1], [1, 1]]
>>> m2 = [[1, 0], [0, 1]]
>>> m1 + m2
[[1, 1], [1, 1], [1, 0], [0, 1]]
```

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

¡No se ha realizado la suma de las matrices, sino la **concatenación** de la listas!

Listas anidadas (o listas de listas)

- Suma de los elementos de una matriz

$$\text{suma} = \sum_{i=0}^{nf-1} \sum_{j=0}^{nc-1} m[i][j]$$

```
# Programa que suma los elementos de una matriz usando listas anidadas
def suma_elem_matriz(m):
    """Suma los elementos de una matriz usando lista de listas
    >>> suma_elem_matriz([[1,1,1],[2,2,2]])
    9
    """
    suma = 0
    nf = len(m)
    nc = len(m[0])
    for i in range(nf):
        for j in range(nc):
            suma += m[i][j]
    return suma

if __name__ == '__main__':
    import doctest
    doctest.testmod(verbose=True)
```

Listas anidadas (o listas de listas)

- ¿Es la matriz simétrica?

```
# Programa que evalúa si una matriz es simétrica (True) o no (False)
def MatEsSimetrica(m):
    """Evalúa si una matriz es simétrica (True) o no (False)
    >>> MatEsSimetrica([[1,2,3],[2,2,4],[3, 4,7]])
    True
    >>> MatEsSimetrica([[1,5,3],[2,2,4],[3, 4,7]])
    False
    """
    for i in range(len(m)):
        for j in range(len(m)):
            if m[i][j] != m[j][i]:
                return False
    return True
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 4 \\ 3 & 4 & 7 \end{bmatrix}$$

Listas anidadas (o listas de listas)

- ¿Matriz es simétrica? (usando while)

```
# Programa que evalúa si una matriz es simétrica (True) o no (False)
def MatEsSimetricaW(m):
    """Evalúa si una matriz es simétrica (True) o no (False)
    >>> MatEsSimetricaW([[1,2,3],[2,2,4],[3, 4,7]])
    True
    >>> MatEsSimetricaW([[1,5,3],[2,2,4],[3, 4,7]])
    False
    """
    sim = True
    i = 0
    while i < len(m) and sim:
        j = 0
        while j < len(m) and sim:
            if m[i][j] != m[j][i] and sim:
                sim = False
            j += 1
        i += 1
    return sim
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 2 & 2 & 4 \\ 3 & 4 & 7 \end{bmatrix}$$

Listas anidadas (o listas de listas)

- Ejemplo de **suma de dos matrices**, usando función-iterador **zip**

```
# Programa que suma los elementos de una matriz
def suma_matrices(m1, m2):
    '''
    >>> suma_matrices([[1,1],[2,2]],[[3,3],[4,4]])
    [[4, 4], [6, 6]]
    '''
    ms = []
    for x1, x2 in zip(m1, m2):
        fila = []
        for y1, y2 in zip(x1, x2):
            fila.append(y1 + y2)
        ms.append(fila)
    return ms

if __name__ == "__main__":
    print(suma_matrices([[1,1],[2,2]],[[3,3],[4,4]]))
    import doctest
    doctest.testmod()
```

$$\begin{bmatrix}1 & 1\\2 & 2\end{bmatrix}+\begin{bmatrix}3 & 3\\4 & 4\end{bmatrix}$$

Listas anidadas (o listas de listas)

Comprensiones de listas (list comprehensions)

- Ejemplo: Crear una matriz nfilas x ncolum pre-asignada de 0's, ejemplo:

$$A=\begin{bmatrix}0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0\end{bmatrix}$$

```
A = [[0]*ncolum for i in range(nfilas)] # List comprehension
```

```
>>> nfilas= 3
>>> ncolum = 4
>>> A = [[0]*ncolum for i in range(nfilas)]
>>> A
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

Listas anidadas (o listas de listas)

Ejemplo **suma**, a través de la composición iterativa for – in (función)

```
def SumaMat(m1,m2):
    """ Funcion que calcula la suma de 2 matrices
    >>> SumaMat([[1, 1], [1, 1]],[[1, 0], [0, 1]])
    [[2, 1], [1, 2]]
    >>> SumaMat([[1, 1, 1], [1, 1, 1]],[[1, 0], [0, 1]])
    -1
    """
    f1 = len(m1)
    c1 = len(m1[0])
    f2 = len(m2)
    c2 = len(m2[0])
    if f1 != f2 or c1 != c2:  # chequea que sean del mismos tamaño
        msuma = -1            # -1 indica: error, matrices distinto tamaño
    else:
        msuma = [[0]*c1 for i in range(f1)] # matriz de 0's
        for i in range(f1):
            for j in range(c1):
                msuma[i][j] = m1[i][j] + m2[i][j]
    return msuma

print(SumaMat([[1,2],[3,4]],[[5,5],[5,5]]))
```

$$\begin{bmatrix}1 & 1\\ 1 & 1\end{bmatrix} + \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix} = \begin{bmatrix}2 & 1\\ 1 & 2\end{bmatrix}$$

Listas anidadas (o listas de listas): Listas de listas heterogéneas

- Listas de listas para guardar formularios o registros de datos.
- Ejemplo. De una **persona** se guarda (i) el nombre, (ii) teléfono y (iii) dirección.
  En una **Agenda** se guardan **personas**

```
>>>  Persona = ["Ana", "610000003", "Aragón 333"]
>>>  Agenda = [["Juan", "610000001", "Balmes 111"],
               ["Pedro", "610000002", "París 222"],
                Persona]
>>> Agenda.sort()
[['Ana', '610000003', 'Aragón 333'],
 ['Juan', '610000001', 'Balmes 111'],
 ['Pedro', '610000002', 'París 222']]
```

Listas anidadas (o listas de listas): Listas de listas heterogéneas

Ejemplo 1) Se dispone de una lista que contiene las ventas de los comerciales de una empresa a lo largo de un mes. Por cada comercial, la lista contiene una sub-lista con el nombre del comercial, seguido de sus ventas.

a.1) Función que devuelva una lista con las tuplas de los nombre y la mayor venta de cada comercial (**opción listas concatenadas**)

```
def ventas_max(lst):
    '''
    >>> l_ventas = [['joan', 389, 200], ['maria', 600, 700, 58], \
    ['pedro', 35, 22], ['sara', 12, 800]]
    >>> ventas_max(l_ventas)
    [('joan', 389), ('maria', 700), ('pedro', 35), ('sara', 800)]
    '''
    l_max = []
    for comercial in lst:
        l_max += [(comercial[0], max(comercial[1:]))]
    return l_max
```

Listas anidadas (o listas de listas): Listas de listas heterogéneas

Ejemplo 1) Se dispone de una lista que contiene las ventas de los comerciales de una empresa a lo largo de un mes. Por cada comercial, la lista contiene una sub-lista con el nombre del comercial, seguido de sus ventas.

a.2) Función que devuelva una lista con las tuplas de los nombre y la mayor venta de cada comercial (**opción añadir a lista con método append**)

```
def ventas_max(lst):
    '''
    >>> l_ventas = [['joan', 389, 200], ['maria', 600, 700, 58], \
    ['pedro', 35, 22], ['sara', 12, 800]]
    >>> ventas_max(l_ventas)
    [('joan', 389), ('maria', 700), ('pedro', 35), ('sara', 800)]
    '''
    l_max = []
    for comercial in lst:
        l_max.append((comercial[0], max(comercial[1:])))
    return l_max
```

Listas anidadas (o listas de listas): Listas de listas heterogéneas

Ejemplo 2) Se dispone de una lista que contiene las ventas de los comerciales de una empresa a lo largo de un mes.

b) Función que devuelva una tupla con el nombre y la cantidad de la mayor suma de ventas

```
def venta_mayor(lst):
    '''
    >>> l_ventas = [['joan', 389, 200], ['maria', 600, 700, 58], \
    ['pedro', 35, 22], ['sara', 12, 800]]
    >>> venta_mayor(l_ventas)
    ('maria', 1358)
    '''
    Nombre = ''
    Mayor = 0
    for comercial in lst:
        SumaVentas = sum(comercial[1:])
        if SumaVentas > Mayor:
            Mayor = SumaVentas
            Nombre = comercial[0]
    return Nombre, Mayor
```

Listas anidadas (o listas de listas): Listas de listas heterogéneas

Ejemplo 3) La información de un jugador de balonmano base se guarda en una lista formada por su nombre (un string), su edad (un entero) y un número variable de posiciones donde puede jugar en el equipo (secuencia de strings).

```
def cuantos(ljug, e, pos):
    n = 0
    for jugador in ljug:
        edat = jugador[1]
        if edat == e:
            lpos = jugador[2:]
            if pos in lpos:
                n = n + 1
    return n
```

```
>>> lista = [['Joan', 13, 'lateral', 'pivot'], ['Marta', 11, 'central', 'lateral'],\
             ['Marc', 11, 'portero'], ['Clara', 12, 'pivot', 'central', 'lateral'],\
             ['Luis', 10, 'portero', 'extremo', 'central', 'pivot'], \
             ['Alberto', 11, 'lateral', 'pivot']]
>>> cuantos(lista,11,'lateral')
2
```

Listas anidadas (o listas de listas)

Ejemplo 4. **Lista de datos de pacientes**. Diseña una función que, dada una lista de datos de pacientes en listas que contienen como elementos "listas" con elementos heterogéneos: *apellido*, *código de la seguridad social* y *edad*, devuelva el **apellido** del paciente con mayor edad.

```
def edad_m_apellido(lst):
    """
    """
    Mayor = lst[0][2]
    Nom = lst[0][0]
    for pacient in lst:
        if pacient[2]>Mayor:
            Mayor = pacient[2]
            Nom = pacient[0]
    return Nom
```

```
>>> paciente1 = ['Puig','c1234', 56]
>>> lista = [paciente1,['Martin','c3456',39],['Lee','c543',45]]
>>> edad_m_apellido(lista)
'Puig'
```