

Funciones y métodos de listas

Las listas en Python vienen con una colección de **operaciones** muy completa. Veamos algunas de ellas:

Rangos o secciones o slices

En Python hay distintos modos de extraer fragmentos de una lista. Imposible explicarlo mejor que con un ejemplo.

In [1]:

```
lista = list(range(10, 101, 10))
print("a) ", lista)
print("b) ", lista[3:6])
print("c) ", lista[:6])
print("d) ", lista[6:])
print("e) ", lista[-1:0:-1])
print("f) ", lista[::-1])
print("g) ", lista[0:10:1])
print("h) ", lista[ : :1])
print("i) ", lista[0:10:2])
print("j) ", lista[::-2])
```

- a) [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
- b) [40, 50, 60]
- c) [10, 20, 30, 40, 50, 60]
- d) [70, 80, 90, 100]
- e) [100, 90, 80, 70, 60, 50, 40, 30, 20]
- f) [100, 90, 80, 70, 60, 50, 40, 30, 20, 10]
- g) [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
- h) [10, 20, 30, 40, 50, 60, 70, 80, 90, 100]
- i) [10, 30, 50, 70, 90]
- j) [100, 80, 60, 40, 20]

Operaciones de uso frecuente

Repasamos seguidamente algunas de las operaciones más frecuentes con listas.

Observa que muchas de ellas trabajan *in place*, esto es, modificando el objeto de referencia.

append

```
lista.append(objeto)
```

Añade un objeto al final de una lista.

In [2]:



```
lista.append(666)
print(lista)
```

```
[10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 666]
```

Ojo: rangos \neq listas

En efecto, un rango no es una lista...

In [3]:



```
lista = range(5)
print(lista)

lista.append(666) # error: es imposible aplicar este método a un rango
```

```
range(0, 5)
```

```
-----
AttributeError                                Traceback (most recent call last)
<ipython-input-3-cea5fffd10b9d> in <module>
      2 print(lista)
      3
----> 4 lista.append(666) # error: es imposible aplicar este método a un ran
go
```

AttributeError: 'range' object has no attribute 'append'

... pero se puede convertir en una lista.

In [4]:



```
lista = range(5)
print(lista)

lista = list(lista)
print(lista)

lista.append(666) # Ahora sí
print(lista)
```

```
range(0, 5)
[0, 1, 2, 3, 4]
[0, 1, 2, 3, 4, 666]
```

count

```
lista.count(valor)
```

Da el número de apariciones de un valor en una lista.

In [5]:



```
lista = [1, 3, 1, 7, 9, 1, 7]
lista.count(1), lista.count(3), lista.count(7), lista.count(100)
```

Out[5]:

```
(3, 1, 2, 0)
```

extend

```
lista.extend(iterable)
```

Extiende una lista, al final, añadiendo una colección de elementos (iterable).

In [6]:



```
lista = [1, 2, 3, 4, 5]
print(lista)
lista.extend(range(3))
print(lista)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, 0, 1, 2]
```

In [7]:



```
# Mira lo que pasaría si hubiéramos usado append:

lista = [1, 2, 3, 4, 5]
print(lista)
lista.append([0, 1, 2])
print(lista)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4, 5, [0, 1, 2]]
```

index

```
lista.index(valor [, start[, stop]]) -> integer
```

Da el primer índice en que se encuentra el valor.

In [8]:



```
lista = [10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
print(lista)
print(lista.index(40), lista.index(40, 5))
```

```
[10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
3 8
```

La operación `index` dispara un `ValueError` si el valor no está presente:

In [9]:

```
print(lista)
print(lista.index(40, 5, 6)) # Dará un error
```

```
[10, 20, 30, 40, 50, 10, 20, 30, 40, 50]
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-9-851547febd3c> in <module>
      1 print(lista)
----> 2 print(lista.index(40, 5, 6)) # Dará un error

ValueError: 40 is not in list
```

insert

```
lista.insert(index, objeto)
```

Inserta un objeto en la posición indicada.

In [10]:

```
lista = [1, 2, 3, 4, 5]
lista.insert(4, 666)
print(lista)
lista.insert(0, 777)
print(lista)
lista.insert(-1, 888)
print(lista)
lista.insert(len(lista), 888)
print(lista)
```

```
[1, 2, 3, 4, 666, 5]
[777, 1, 2, 3, 4, 666, 5]
[777, 1, 2, 3, 4, 666, 888, 5]
[777, 1, 2, 3, 4, 666, 888, 5, 888]
```

pop

```
lista.pop([index]) -> item
```

Elimina (y devuelve) el elemento en la posición dada por index. Por defecto, el elemento será el último.

In [11]:



```
lista = [1, 2, 3, 4, 5]
print(lista)
lista.pop()
print(lista)
lista.pop(1)
print(lista)
```

```
[1, 2, 3, 4, 5]
[1, 2, 3, 4]
[1, 3, 4]
```

Además de modificar la lista, esta operación devuelve el elemento eliminado:

In [12]:



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

x = lista.pop()

print(x)
print(lista)
```

```
[1, 2, 3, 4, 5]
5
[1, 2, 3, 4]
```

La operación `pop` dispara un `IndexError` si la lista está vacía o el índice está fuera de rango:

In [13]:



```
lista = [1, 2, 3]
lista.pop(10)
```

```
-----
IndexError                                Traceback (most recent call last)
<ipython-input-13-8de469789510> in <module>
      1 lista = [1, 2, 3]
----> 2 lista.pop(10)
```

IndexError: pop index out of range

remove

```
lista.remove(valor)
```

Elimina el primer elemento igual al valor dado. Dispara un `ValueError` si el valor no está presente.

In [14]:



```
lista = [1, 2, 3, 4, 5, 3, 3, 3]
print(lista)
lista.remove(3)
print(lista)
```

```
[1, 2, 3, 4, 5, 3, 3, 3]
[1, 2, 4, 5, 3, 3, 3]
```

reverse

```
lista.reverse()
```

Invierte una lista *in place*

In [15]:



```
lista = [1, 2, 3, 4, 5]
print(lista)
lista.reverse()
print(lista)
```

```
[1, 2, 3, 4, 5]
[5, 4, 3, 2, 1]
```

sort

```
lista.sort(key=None, reverse=False)
```

Ordena una lista *in place*

In [16]:



```
lista = [13, 4586, 1, 989, 34]
print(lista)

lista.sort()
print(lista)

lista.sort(key=lambda n: n%10) # ordena atendiendo a la última cifra
print(lista)

lista.sort(key=lambda n: n%10, reverse=True) # ordena, según la última cifra, de mayor a menor
print(lista)
```

```
[13, 4586, 1, 989, 34]
[1, 13, 34, 989, 4586]
[1, 13, 34, 4586, 989]
[989, 4586, 34, 13, 1]
```

In [17]:



```
# Podemos ordenar en una lista nueva, esto es, no in place:

lista = [13, 4586, 1, 989, 34]
print(lista)

lista2 = sorted(lista)
print(lista)
print(lista2)

# La función sorted admite también los parámetros por defecto:

lista2 = sorted(lista, key=lambda n: n%2) # pares, impares
print(lista)
print(lista2)
```

```
[13, 4586, 1, 989, 34]
[13, 4586, 1, 989, 34]
[1, 13, 34, 989, 4586]
[13, 4586, 1, 989, 34]
[4586, 34, 13, 1, 989]
```

El método sort es muy eficiente y muy adaptable a necesidades muy variadas, usando adecuadamente los parámetros por defecto:

In [18]:



```
lista = [("Manuela", "Gonzalo"), ("Manuela", "Donato"), ("Juan", "Pérez"), ("Alba", "Jimeno")

def clave(par):
    nombre, apellido = par
    return nombre + apellido

print(lista)

lista.sort(key=clave)

print(lista)
```

```
[('Manuela', 'Gonzalo'), ('Manuela', 'Donato'), ('Juan', 'Pérez'), ('Alba', 'Jimeno'), ('Alba', 'Zaragoza')]
[('Alba', 'Jimeno'), ('Alba', 'Zaragoza'), ('Juan', 'Pérez'), ('Manuela', 'Donato'), ('Manuela', 'Gonzalo')]
```

Concatenación de listas

In [19]:

```

lista1 = list(range(3))
lista2 = lista1 + [7, 8, 9] # La concatenación crea una lista nueva
print(lista1, lista2)
lista2[1] = 666
print(lista1, lista2)

```

```

[0, 1, 2] [0, 1, 2, 7, 8, 9]
[0, 1, 2] [0, 666, 2, 7, 8, 9]

```

Ejemplo 1. Posiciones de elementos

In [20]:

```

def posic(lst, elem):
    """
    Da la primera posición de un elemento en una lista
    Si el elemento no está en la lista, el resultado es -1

    Parameters
    -----
    elem: Alpha, un tipo cualquiera
    lst: lista de Alpha

    Returns
    -----
    pos si, si elem está presente en lst[pos], y no antes
    -1, en caso contrario

    Example
    -----
    >>> posic([1, 2, 5, 45], 5)
    2
    """
    i = 0
    while i < len(lst) and lst[i] != elem:
        i += 1
    if i == len(lst):
        return -1
    else:
        return i

```

In [21]:

```

l = list(range(5,15))
print(l)
print(posic(l, 7), posic(l, 20))

```

```

[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
2 -1

```


In [22]:



```
print(1)
print(l.index(8)) # En realidad, existe un método index predefinido :-)
```

```
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
3
```

In [23]:



```
1
l.index(15) # pero si el elemento no está, el método index da un error
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-23-c05323e57243> in <module>
      1 l
----> 2 l.index(15) # pero si el elemento no está, el método index da un error
or

ValueError: 15 is not in list
```

Ejemplo 2. Eliminación de elementos en una lista

In [24]:



```
def elim(lst, ini, end):
    lst = lst[:ini] + lst[end:]

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(lista)

elim(lista, 3, 5)
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [25]:



```
# Recuerda: tenemos una operación pra eliminar un elemento de una posición...

lista = [5, 6, 7, 8, 9, 10, 11, 12, 13]
print(lista)
lista.pop()
print(lista)
```

```
[5, 6, 7, 8, 9, 10, 11, 12, 13]
[5, 6, 7, 8, 9, 10, 11, 12]
```

Diseñemos una función que elimina todos los elementos de una lista entre dos posiciones dadas:

In [26]:



```
def eliminar(lst, ini, end):
    """
    Eliminación *in place* de todos los elementos lst[j], para ini <= j < end

    Parameters
    -----
    lst: [x]
    ini: int
    end: int

    0 <= ini < end <= len(lst)

    Returns
    -----
    None

    Action
    -----
    modifica lst, eliminando todos los elementos entre ini y end

    Example
    -----
    >>> l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    >>> eliminar(l, 3, 5)
    >>> print(l)
    [1, 2, 3, 6, 7, 8, 9, 10]
    """

    # Primero adelantamos todos los elementos tras la franja que se va a eliminar,
    # esto es, movemos los elementos lst[i], para end <= i < len(lst),
    # a las posiciones que empiezan en ini:

    pos_end = end
    pos_ini = ini
    while pos_end < len(lst):
        lst[pos_ini] = lst[pos_end]
        pos_ini += 1
        pos_end += 1

    # Ahora, eliminamos los end-ini elementos desde el final de la lista:
    for i in range(end-ini):
        lst.pop()
```

In [27]:



```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(lista)

eliminar(lista, 3, 5)
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 3, 6, 7, 8, 9, 10]
```

In [28]:

```
def elim(lst, ini, end):  
    lst = lst[:ini] + lst[end:]  
  
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
print(lista)  
  
elim(lista, 3, 5)  
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [29]:

```
# No funciona como se esperaba...  
# porque la modificación del parámetro lst sólo se realiza localmente. Veámoslo:  
  
def elim(lst, ini, end):  
    lst = lst[:ini] + lst[end:]  
    print("local: ", lst)  
  
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
print(lista)  
  
elim(lista, 3, 5)  
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
local: [1, 2, 3, 6, 7, 8, 9, 10]  
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

In [30]:

```
# Pero podríamos haberlo hecho:  
  
def elim(lst, ini, end):  
    lst[:] = lst[:ini] + lst[end:]  
  
# Observa la diferencia entre lst = ... y lst[:] = ...  
  
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
print(lista)  
  
elim(lista, 3, 5)  
print(lista)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  
[1, 2, 3, 6, 7, 8, 9, 10]
```

Como ya podíamos imaginar, una función tan útil como ésta ya está predefinida:

In [31]:



```
lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
print(lista)

del lista[2]
print(lista)

del lista[2:5] # Borra los elems. entre las posics. 2 y 5 (excluida), como delete(lista, 2,
print(1)
```

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
[1, 2, 4, 5, 6, 7, 8, 9, 10]
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

In [32]:



```
del lista[2:]
lista
```

Out[32]:

```
[1, 2]
```

In [33]:



```
del lista[5:2] # queda sin efecto, al tratarse de un rango vacío
lista
```

Out[33]:

```
[1, 2]
```

Ejemplo 3. Inversión de una lista

In [34]:



```
def invierte(lst):
    """
    Permuta *in place* los elementos de la lista dada,
    dejándola en orden inverso al proporcionado

    Parameters
    -----
    lst: [x]

    Returns
    -----
    None

    Example
    -----
    >>> lista = [1, 2, 5, 4]
    >>> invierte(lista)
    >>> lista

    """
    # Permutamos la primera mitad de los elementos de la lista
    # con los de la otra mitad
    for i in range(0, len(lst)//2):
        j = len(lst) - 1 - i
        lst[i], lst[j] = lst[j], lst[i]
```

In [35]:



```
lista = list(range(10))
print(lista)
invertir(lista)
print(lista)

# Cuidado: esta función NO devuelve valor alguno: únicamente permuta los elementos *in place*
print(invertir(lista))
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
None
```

Por supuesto, esta función también está definida:

In [36]:



```
l = list(range(10))
print(l)
l.reverse()
print(l)
```

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

enumerate

Utilísima es la siguiente función. Surge con mucha frecuencia.

In [37]:

```
list(enumerate("aeiou"))
```

Out[37]:

```
[(0, 'a'), (1, 'e'), (2, 'i'), (3, 'o'), (4, 'u')]
```

Catálogo de funciones predefinidas sobre listas

In [38]:

```
help(list)
```

Help on class list in module builtins:

```
class list(object)
| list(iterable=(), /)
|
| Built-in mutable sequence.
|
| If no argument is given, the constructor creates a new empty list.
| The argument must be an iterable if specified.
|
| Methods defined here:
|
| __add__(self, value, /)
|     Return self+value.
|
| __contains__(self, key, /)
|     Return key in self.
|
| delitem (self, key, /)
```