Listas

Definición y características principales

Las listas son conjuntos ordenados de elementos (números, cadenas, listas, etc). Las listas se delimitan por corchetes (`[]`) y los elementos se separan por comas.

Las listas pueden contener elementos del mismo tipo:

```
>>> primos = [2, 3, 5, 7, 11, 13]
>>> diasLaborables = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes"]
```

O pueden contener elementos de tipos distintos:

```
>>> fecha = ["Lunes", 27, "Octubre", 1997]
```

O pueden contener otras listas:

```
>>> peliculas = [["Senderos de Gloria", 1957], ["Hannah y sus hermanas", 1986]]
```

Las listas pueden tener muchos niveles de anidamiento:

```
>>> directores = [["Stanley Kubrick", ["Senderos de Gloria", 1957]], ["Woody
Allen", ["Hannah y sus hermanas", 1986]]]
```

Las variables de tipo lista hacen referencia a la lista completa.

```
>>> lista = [1, "a", 45]
>>> lista
[1, 'a', 45]
```

Una lista que no contiene ningún elemento se denomina **lista vacía**:

```
>>> lista = []
>>> lista
[]
```

Usando otras variables en una lista

Al definir una lista se puede hacer referencia a otras variables.

```
>>> nombre = "Pepe"
>>> edad = 25
>>> lista = [nombre, edad]
>>> lista
['Pepe', 25]
```

Como siempre, hay que tener cuidado al modificar una variable que se ha utilizado para definir otras variables, porque esto puede afectar al resto de variables:

- Si se trata **objetos inmutables**, el resto de variables no resultan afectadas, como muestra el siguiente ejemplo:

```
>>> nombre = "Pepe"
>>> edad = 25
>>> lista = [nombre, edad]
>>> lista
['Pepe', 25]
>>> nombre = "Juan"
>>> lista
['Pepe', 25]
```

- Pero si se trata de **objetos mutables** y al modificar la variable se modifica el objeto, el resto de variables sí resultan afectadas, como muestra el siguiente ejemplo:

```
>>> nombres = ["Ana", "Bernardo"]
>>> edades = [22, 21]
>>> lista = [nombres, edades]
>>> lista
[['Ana', 'Bernardo'], [22, 21]]
>>> nombres += ["Cristina"]
>>> lista
[['Ana', 'Bernardo', 'Cristina'], [22, 21]]
```

Conversión

Podemos convertir otros tipos de datos en una lista usando la función `list()`

```
>>> list('Python mola')
['P', 'y', 't', 'h', 'o', 'n', ' ', 'm', 'o', 'l', 'a']
```

También podemos utilizarlo para crear una lista vacia:

```
>>> list()
[]
```

Concatenación de listas

Las listas se pueden concatenar con el símbolo de la suma (+):

```
>>> vocales = ["E", "I", "O"]
>>> vocales
['E', 'I', 'O']
>>> vocales = vocales + ["U"]
>>> vocales
['E', 'I', 'O', 'U']
>>> vocales = ["A"] + vocales
>>> vocales
['A', 'E', 'I', 'O', 'U']
```

El operador suma (+) necesita que los dos **operandos** sean listas:

```
>>> vocales = ["E", "I", "O"]
>>> vocales = vocales + "Y"
Traceback (most recent call last):
  File "<pyshell#2>", line 1, in <module>
    vocales = vocales + "Y"
  TypeError: can only concatenate list (not "str") to list
```

También se puede utilizar el operador `+=` para añadir elementos a una lista:

```
>>> vocales = ["A"]
>>> vocales += ["E"]
>>> vocales
['A', 'E']
```

Aunque en estos ejemplos, los operadores + y += den el mismo resultado, no son equivalentes, como se explica en la lección de Variables 2.

Añadir al final de una lista

Podemos añadir elementos al final de la lista usando la concatenación como acabamos de ver o también usando el método `append()`

```
>>> vocales = ["A", "E", "I", "O"]
>>> vocales.append("U")
```

Fíjate que en este caso se añade un elemento, no una lista.

Manipular elementos individuales de una lista

Cada elemento se identifica por su **posición** en la lista, teniendo en cuenta que se empieza a contar por **0**. A las posiciones de los elementos de la lísta se les llama **índices**

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[0]             # Mostramos el valor de la lista con índice 0
27
>>> fecha[1]
Octubre
>>> fecha[2]
1997
```

Se pueden utilizar **números negativos** (el último elemento tiene el índice -1 y los elementos anteriores tienen valores descendentes):

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[-1]
1997
>>> fecha[-2]
Octubre
>>> fecha[-3]
27
```

No se puede hacer referencia a elementos fuera de la lista:

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[3]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fecha[3]
  Index error: list index out of range
>>> fecha[-4]
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    lista[-4]
IndexError: list index out of range
```

Se puede **modificar** cualquier elemento de una lista haciendo referencia a su posición:

```
>>> fecha = [27, "Octubre", 1997]
>>> fecha[2] = 1998
>>> fecha[0]
27
>>> fecha[1]
Octubre
>>> fecha[2]
1998
```

Insertar en cualquier posición de una lista

Si queremos insertar un elemento en cualquier posición de una lista debemos usar `insert()` especificando el índice del nuevo elemento:

```
>>> vocales = ["A", "E", "O", "U"]
>>> vocales.insert(2, "I")
>>> vocales
['A', 'E', 'I', 'O', 'U']
```

Manipular sublistas

De una lista se pueden generar **sublistas**, utilizando la notación `nombre_lista[inicio:límite]`, donde inicio y límite hacen el mismo papel que en el tipo `range(inicio, límite)`.

```
>>> dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"]
>>> dias[1:4]   # Se muestra una lista con los elementos con índices 1, 2 y 3
['Martes', 'Miércoles', 'Jueves']
>>> dias[4:5]   # Se muestra una lista con el elemento con índice 4
['Viernes']
>>> dias[4:4]  # Se muestra una lista vacía
[]
>>> dias[:4]  # Se muestra una lista con los índices 0, 1, 2 y 3
['Lunes', 'Martes', 'Miércoles', 'Jueves']
>>> dias[4:]  # Se muestra una lista desde el índice 4 (incluido) en adelante
['Viernes', 'Sábado', 'Domingo']
>>> dias[:]   # Se muestra una lista con todos los valores
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
```

También se pueden generar sublistas usando un tercer operador, que al igual que en `range` indica el incremento a aplicacar al recorrer la lista:

```
>>> dias = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes", "Sábado",
"Domingo"]
>>> dias[0:7:2]   # Se muestra una lista con los elementos con índices 0, 2, 4 y
6
['Lunes', 'Miércoles', 'Viernes', 'Domingo']
>>> dias[:5:3]
['Lunes', 'Jueves']
>>> dias[-1:4:-1]  # Se puede extraer en orden inverso
['Domingo', 'Sábado']
>>> dias[-1:0:-1]
['Domingo', 'Sábado', 'Viernes', 'Jueves', 'Miércoles', 'Martes']
>>> dias[-1:0:-1]
['Domingo', 'Sábado', 'Viernes', 'Jueves', 'Miércoles', 'Martes']
```

Se puede **modificar** una lista modificando **sublistas**. De esta manera se puede modificar un elemento o varios a la vez e **insertar** o **eliminar** elementos.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> letras[1:4] = ["X"]      # Se sustituye la sublista ['B','C','D'] por ['X']
>>> letras
['A', 'X', 'E', 'F', 'G', 'H']
>>> letras[1:4] = ["Y", "Z"] # Se sustituye la sublista ['X','E','F'] por
['Y','Z']
>>> letras
['A','Y', 'Z', 'G', 'H']
>>> letras[0:1] = ["Q"]      # Se sustituye la sublista ['A'] por ['Q']
>>> letras
['Q', 'Y', 'Z', 'G', 'H']
>>> letras[3:3] = ["U", "V"] # Inserta la lista ['U','V'] en la posición 3
>>> letras
['Q', 'Y', 'Z', 'U', 'V', 'G', 'H']
>>> letras[0:3] = []         # Elimina la sublista ['Q','Y', 'Z']
>>> letras
['U', 'V', 'G', 'H']
```

Al definir sublistas, Python acepta valores fuera del rango, que se interpretan como **extremos** (al final o al principio de la lista).

```
>>> letras = ["D", "E", "F"]
>>> letras[3:3] = ["G", "H"]          # Añade ["G", "H"] al final de la lista
>>> letras
['D', 'E', 'F', 'G', 'H']
>>> letras[100:100] = ["I", "J"]      # Añade ["I", "J"] al final de la lista
>>> letras
['D', 'E', 'F', 'G', 'H', 'I', 'J']
>>> letras[-100:-50] = ["A", "B", "C"] # Añade ["A", "B", "C"] al principio de la
lista
>>> letras
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J']
```

La palabra reservada del

La palabra reservada `del` permite eliminar un elemento o varios elementos a la vez de una lista, e incluso la misma lista.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[4]   # Elimina el elemento con ínide 4 de la lista
>>> letras
['A', 'B', 'C', 'D', 'F', 'G', 'H']
>>> del letras[1:4] # Elimina la sublista ['B', 'C', 'D']
>>> letras
['A', 'F', 'G', 'H']
>>> del letras      # Elimina completamente la lista
>>> letras
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in ?
    letras
  NameError: name 'letras' is not defined
```

Si se intenta borrar un elemento que no existe, se produce un error:

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[10]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    del letras[10]
IndexError: list assignment index out of range
```

Aunque si se hace referencia a **sublistas**, Python sí que acepta valores fuera de rango, pero lógicamente no se modifican las listas.

```
>>> letras = ["A", "B", "C", "D", "E", "F", "G", "H"]
>>> del letras[100:200] # No elimina nada
>>> letras
['A', 'B', 'C', 'D', 'E', 'F', 'G', 'H']
```

Copiar una lista

Con variables de tipo **entero**, **decimal** o de **cadena**, es fácil tener una copia de una variable para conservar un valor que en la variable original se ha perdido:

```
>>> a = 5
>>> b = a # Hacemos una copia del valor de a
>>> a, b
(5, 5)
>>> a = 4 # de manera que aunque cambiemos el valor de a ...
>>> a, b  # ... b conserva el valor anterior de a en caso de necesitarlo
(4, 5)
```

Pero si hacemos esto mismo con listas, nos podemos llevar un sorpresa:

```
>>> lista1 = ["A", "B", "C"]
>>> lista2 = lista1 # Intentamos hacer una copia de la lista lista1
>>> lista1, lista2
(['A', 'B', ,'C'] ['A', 'B', ,'C'])
>>> del lista1[1]   # Eliminamos el elemento ['B'] de la lista lista1 ...
>>> lista1, lista2  # ... pero descubrimos que también ha desaparecido de la
lista lista2
(['A', 'C'] ['A', 'C'])
```

El motivo de este comportamiento, es que los enteros, decimales y cadenas son objetos **inmutables** y las listas son objetos **mutables**. Cuando hacemos `lista2 = lista1` los dos objetos apuntan a la misma dirección de memoria (el contenido de la lista está en la misma ubicación)

Copiar listas usando sublistas

Si queremos copiar una lista, de manera que conservemos su valor aunque modifiquemos la lista original debemos utilizar la notación de **sublistas**.

```
>>> lista1 = ["A", "B", "C"]
>>> lista2 = lista1[:] # Hacemos una copia de la lista lista1
>>> lista1, lista2
(['A', 'B', 'C'] ['A', 'B', 'C'])
>>> del lista1[1]      # Eliminamos el elemento ['B'] de la lista lista1 ...
>>> lista1, lista2     # ... y en este caso lista2 sigue conservando el valor
original de lista1
(['A', 'C'] ['A', 'B', 'C'])
```

En este caso, al hacer `lista2 = lista1[:]` se genera una nueva lista que se almacena en otra posición de memnoria y que contiene los mismos elementos que los de lista1. Por tanto, `lista1` y `lista2` hacen referencia a listas distintas (aunque tengan los mismos valores, están almacenadas en lugares distintos de la memoria del ordenador). Por eso, al eliminar un elemento de `lista1`, no se elimina en `lista2`.

Puedes ver mejor lo que ocurre ejecutando paso a paso el siguiente código en Pythontutor

Copiar listas usando la función list()

También podemos hacer una copia de una lista usando el operador **list** que, cómo vimos, nos permite convertir cualquier elemento **iterable** en una lista. En particular, si se lo aplicacamos a una lista nos genera una copia de la misma y la almacena en otra posición de memoria:

```
>>> dias = ["Lunes", "Martes", "Miércoles"]
>>> dias2 = list(dias)
>>> dias
['Lunes', 'Martes', 'Miércoles']
>>> dias2
['Lunes', 'Martes', 'Miércoles']
>>> del dias[0]
>>> dias
['Martes', 'Miércoles']
>>> dias2
['Lunes', 'Martes', 'Miércoles']
```

Copiar listas mediante copy()

El método `copy()` también permite realizar una copia **dura** de una lista, de forma que las modificaciones en una de las listas no se reflejen en la segunda:

```
>>> vocales = ["A", "E", "I", "O", "U"]
>>> vocales1 = vocales
>>> vocales2 = vocales.copy()
>>> del vocales[0]
>>> vocales
['E', 'I', 'O', 'U']
>>> vocales1
['E', 'I', 'O', 'U']
>>> vocales2
['A', 'E', 'I', 'O', 'U']
```

Recorrer una lista

Mostrar elementos

Se puede recorrer una lista de principio a fin de **dos formas** distintas:

- Una forma es recorrer directamente los elementos de la lista, es decir, que la **variable de control** del bucle tome los valores de la lista que estamos recorriendo:

```
  letras = ["A", "B", "C"]
  for i in letras:
      print(i, end=" ")
```

```
A B C
```

- La otra forma es recorrer **indirectamente** los elementos de la lista, es decir, que la variable de control del bucle tome como valores los **índices** de la lista que estamos recorriendo (0,1 ,2 , etc.). En este caso, para acceder a los valores de la lista hay que utilizar `letras[i]`:

```
letras = ["A", "B", "C"]
l_lista = len(letras)
for i in range(l_lista):
    print(letras[i], end=" ")
```

```
A B C
```

La primera forma es más sencilla, pero sólo permite recorrer la lista de principio a fin y utilizar los valores de la lista.

La segunda forma es un poco más complicada, pero permite más flexibilidad, como muestran los siguientes ejemplos:

- Recorrer una lista **al revés**

```
letras = ["A", "B", "C"]
l_lista = len(letras)
for i in range(l_lista - 1, -1, -1):
    print(letras[i], end=" ")
```

```
C B A
```

Modificar los elementos de una lista

Podemos recorrer y al mismo tiempo modificar los elementos de una lisa

```
letras = ["A", "B", "C"]
print(letras)
l_lista = len(letras)
for i in range(l_lista):
    letras[i] = "X"
    print(letras)
```

```
['A', 'B', 'C']
['X', 'B', 'C']
['X', 'X', 'C']
['X', 'X', 'X']
```

Eliminar elementos de la lista

Para eliminar los elementos de una lista necesitamos recorrer la lista al revés. Si recorremos la lista **de principio a fin**, al eliminar un valor de la lista, la **lista se acorta** y cuando intentamos acceder a los **últimos valores** se produce un error de índice fuera de rango, como muestra el siguiente ejemplo en el que se eliminan los valores de una lista que valen "B":

Eliminar valores de una lista (incorrecto)

```
letras = ["A", "B", "C"]
print(letras)
l_lista = len(letras)
for i in range(l_lista):
    if letras[i] == "B":
        del letras[i]
    print(letras)
```

```
  ```python
  ['A', 'B', 'C']
  ['A', 'B', 'C']
  ['A', 'C']
  Traceback (most recent call last):
    File "ejemplo.py", line 4, in <module>
      if letras[i] == "B":
  IndexError: list index out of range
```

La solución es recorrer la lista en orden inverso, de manera que aunque se eliminen elementos y la lista se acorte, los valores que todavía no se han recorrido siguen existiendo en la misma posición que al principio.
```

Eliminar valores de una lista (correcto)

```
  letras = ["A", "B", "C"]
  l_lista = len(letras)
  print(letras)
  for i in range(l_lista -1, -1, -1):
      if letras[i] == "B":
          del letras[i]
      print(letras)
```

```
['A', 'B', 'C']
['A', 'B', 'C']
['A', 'C']
['A', 'C']
```

Saber si un valor está o no en una lista

Para saber si un valor está en una lista se puede utilizar el operador lógico `in`. La sintaxis sería `elemento in lista` y devuelve un valor lógico: `True` si el elemento está en la lista o `False` si el elemento **no** está en la lista.

Por ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:

```
personas_autorizadas = ["Alberto", "Carmen"]
nombre = input("Dígame su nombre: ")
if nombre in personas_autorizadas:
  print("Está autorizado")
else:
  print("No está autorizado")
```

Para saber si un valor **no** está en una lista se pueden utilizar los operadores `not in`. La sintaxis sería `elemento not in lista` y devuelve también un valor lógico: `True` si el elemento **no** está en la lista, `False` si el elemento está en la lista.

Por ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:

```
personas_autorizadas = ["Alberto", "Carmen"]
nombre = input("Dígame su nombre: ")
if nombre not in personas_autorizadas:
  print("No está autorizado")
else:
  print("Está autorizado")
```

Convertir lista a cadena de texto

Dada una lista, podemos convertirla a una cadena de texto, uniendo todos sus elementos mediante algún separador. Para ello hacemos uso del método `join()`. La sintáxis es:

```
<separador>.join(<lista>)
```

Ejemplos:

```
>>> estuche = ['lápiz', 'goma', 'afilador', 'regla']
>>> ', '.join(estuche)
'lápiz, goma, afilador, regla'
>>> ' '.join(estuche)
'lápiz goma afilador regla'
>>> ' | '.join(estuche)
'lápiz | goma | afilador | regla'
```

Métodos de listas

append()

- Vimos como añadir por el final concatenando con operador `+`
- También usando el método `append()`

```
>>> vocales = ["A", "E", "I", "O"]
>>> vocales.append("U")
```

- Fíjate que en este caso se **añade un elemento** , no una lista.

Insert()

- Permite insertar en cualquier posición un elemento
- Se le pasan el índice y el nuevo elemento

```
>>> vocales = ["A", "E", "O", "U"]
>>> vocales.insert(2, "I")
>>> vocales
['A', 'E', 'I', 'O', 'U']
```

copy()

- Permite realizar una copia **dura** de una lista

```
>>> vocales = ["A", "E", "I", "O", "U"]
>>> vocales1 = vocales
>>> vocales2 = vocales.copy()
>>> del vocales[0]
>>> vocales
['E', 'I', 'O', 'U']
>>> vocales1
['E', 'I', 'O', 'U']
>>> vocales2
['A', 'E', 'I', 'O', 'U']
```

join()

- Permite convertir lista en cadena de texto
- Se unen mediante separador
- Sintaxis:

```
<separador>.join(<lista>)
```

Ejemplos:

```
>>> estuche = ['lápiz', 'goma', 'afilador', 'regla']
>>> ', '.join(estuche)
'lápiz, goma, afilador, regla'
>>> ' '.join(estuche)
'lápiz goma afilador regla'
>>> ' | '.join(estuche)
'lápiz | goma | afilador | regla'
```

clear()

- Elimina todos los elementos de una lista
- El resultado es una lista vacia

```
>>> estuche = ['lápiz', 'goma', 'afilador', 'regla']
>>> estuche.clear()
>>> estuche
[]
```

index()

- Permite encontrar el índice de un elemento en una lista
- Si el elemento no existe obtenemos error
- Si el elemento se repite solo devuelve índice primera aparición

```
>>> estuche = ['lápiz', 'goma', 'afilador', 'regla', 'goma']
>>> estuche.index('afilador')
2
>>> estuche.index('papel')
Traceback (most recent call last):
  File "<input>", line 1, in <module>
    estuche.index('papel')
ValueError: 'papel' is not in list

>>> estuche.index('goma')
1
```

count()

- Permite contar cuantas veces aparece un valor en una lista

```
>>> estuche = ['lápiz', 'goma', 'afilador', 'regla', 'goma']
>>> estuche.count('regla')
1
>>> estuche.count('goma')
2
>>> estuche.count('papel')
0
```

Ordenar listas

Python ofrece dos formas de ordenar una lista

Función sorted()

- Usando la función `sorted()`.
    - No se modifica la lista original
    - Devuelve una copia de la lista ya ordenada

```
>>> estuche = ['lápiz', 'goma', 'afilador', 'regla']
>>> estuche2 = sorted(estuche)
>>> estuche
['lápiz', 'goma', 'afilador', 'regla']
>>> estuche2
['afilador', 'goma', 'lápiz', 'regla'
```

Método sort()

- Usando el método sort()
    - Ordena la lista original
    - No devuelte nada al aplicarlo

```
>>> estuche = ['lápiz', 'goma', 'afilador', 'regla']
>>> estuche2 = estuche.sort()
>>> estuche2
>>> estuche
['afilador', 'goma', 'lápiz', 'regla']
```

Funciones matemáticas

- Python ofrece funciones matemáticas a aplicar sobre listas
- Entre ellas: `sum()`, `min()` y `max()`

```
>>> nums = [5, 7, 9, 1, 0, -1, 8]
>>> sum(nums)
29
>>> min(nums)
-1
>>> max(nums)
9
```

Referencias

- Apuntes generados a partir del curso Introducción a la programación con Python que se distribuye bajo una Licencia Creative Commons Reconocimiento-CompartirIgual 4.0 Internacional (CC BY-SA 4.0).

tags: `pro` `ut1` `python` `bucles` `for`