Listas

Las listas son una estructura de datos muy flexible. Python permite manipular listas de muchas maneras. 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:	O pueden contener elementos de tipos distintos:	O pueden contener listas:	Las listas pueden tener muchos niveles de anidamiento:
>>> primos = [2, 3, 5, 7, 11, 13] >>> diasLaborables = ["Lunes", "Martes", "Miércoles", "Jueves", "Viernes"]	>>> fecha = ["Lunes", 27, "Octubre", 1997]	>>> peliculas = [["Senderos de Gloria", 1957], ["Hannah y sus hermanas", 1986]]	>>> 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.	Una lista que no contiene ningún elemento se denomina **lista vacía**:	Al definir una lista se puede hacer referencia a otras variables.

>>> lista = [1, "a", 45]
>>> lista
[1, 'a', 45]
	>>> lista = [ ]
	>>> lista
	[ ]
		>>> nombre = "Pepe"
		>>> edad = 25
		>>> lista = [nombre, edad]
		>>> lista
		['Pepe', 25]

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

Si el contenido se trata de objetos inmutables, no resulta afectado, como muestra el siguiente ejemplo:	Pero si se trata de objetos mutables al modificar la variable se modifica el objeto, como muestra el siguiente ejemplo:

>>> nombre = "Pepe"
>>> edad = 25
>>> lista = [nombre, edad]
>>> lista
['Pepe', 25]
>>> nombre = "Juan"
>>> lista
['Pepe', 25]
	>>> nombres = ["Ana", "Bernardo"]
	>>> edades = [22, 21]
	>>> lista = [nombres, edades]
	>>> lista
	[['Ana', 'Bernardo'], [22, 21]]
	>>> nombres += ["Cristina"]
	>>> lista
	[['Ana', 'Bernardo', 'Cristina'], [22, 21]]

Una lista puede contener listas (que a su vez pueden contener listas, que a su vez etc.):	Se puede acceder a cualquier elemento de una lista escribiendo el nombre de la lista y entre corchetes el número de orden en la lista. El primer elemento de la lista es el número 0.

>>> persona1 = ["Ana", 25]
>>> persona2 = ["Benito", 23]
>>> lista = [persona1, persona2]
>>> lista
[['Ana', 25], ['Benito', 23]]
	>>> lista = [10, 20, 30, 40]
	>>> lista[2]
	30
	>>> lista[0]
	10

Concatenar listas

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

```
>>> lista1 = [10, 20, 30, 40]
>>> lista2 = [30, 20]
>>> lista = lista1 + lista2 + lista1
>>> lista
[10, 20, 30, 40, 30, 20, 10, 20, 30, 40]
```

```
>>> 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 2 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 agregar elementos a una lista:

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

Manipular elementos individuales de una lista

Cada elemento se identifica por su posición en la lista, teniendo en cuenta que comienzan con índice 0.

```
>>> fecha = [2, "Octubre", 1990]
>>> fecha[0]
2
>>> fecha[1]
Octubre
>>> fecha[2]
1990
```

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

```
>>> fecha = [2, "Octubre", 1900]
>>> fecha[3]
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    fecha[3]
  Index error: list index out of range
```

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

```
>>> fecha = [2, "Octubre", 1990]
>>> fecha[-1]
1990
>>> fecha[-2]
Octubre
>>> fecha[-3]
2
```

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

```
>>> fecha = [2, "Octubre", 1990]
>>> fecha[2] = 2000
>>> fecha[0]
2
>>> fecha[1]
Octubre
>>> fecha[2]
2000
```

Manipular sublistas

De una lista se pueden extraer sublistas, utilizando la notación nombreDeLista**[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 extrae una lista con los valores 1, 2 y 3
['Martes', 'Miércoles', 'Jueves']
```

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
```

```
>>> dias[4:5] # Se extrae una lista con el valor 4
['Viernes']
>>> dias[4:4] # Se extrae una lista vacía
[]
>>> dias[:4]  # Se extrae una lista hasta el valor 4 (no incluido)
['Lunes', 'Martes', 'Miércoles', 'Jueves']
>>> dias[4:]  # Se extrae una lista desde el valor 4 (incluido)
['Viernes', 'Sábado', 'Domingo']
>>> dias[:]   # Se extrae una lista con todos los valores
['Lunes', 'Martes', 'Miércoles', 'Jueves', 'Viernes', 'Sábado', 'Domingo']
```

```
['A', 'X', 'E', 'F', 'G', 'H']
>>> letras[1:4] = ["Y", "Z"] # Se sustituye la sublista ['X','E','F'] por ['Y','Z']
['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 la sublista ['E']
>>> 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:	Pero si hacemos esto mismo con listas, nos podemos llevar un sorpresa:

>>> 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)
	>>> 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. 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 el primer caso las variables *lista1* y *lista2* hacen referencia a la misma lista almacenada en la memoria. Por eso al eliminar un elemento de *lista1*, también desaparece de *lista2*.

Sin embargo en el segundo caso *lista1* y *lista2* hacen referencia a listas distintas (aunque tengan los mismos valores, están almacenadas en lugares distintos de la memoria). Por eso, al eliminar un elemento de *lista1*, no se elimina en *lista2*.

Recorrer una lista

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:	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]:
Recorrer una lista directamente	**Recorrer una lista indirectamente**

letras = ["A", "B", "C"]
for i in letras:
  print(i, end=" ")
A B C
	letras = ["A", "B", "C"]
	for i in range(len(letras)):
	  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 más complicada, pero permite más flexibilidad, como muestran los siguientes ejemplos:

Recorrer una lista al revés	Recorrer y modificar una lista

letras = ["A", "B", "C"]
for i in range(len(letras)-1, -1, -1):
  print(letras[i], end=" ")
C B A
	letras = ["A", "B", "C"]
	print(letras)
	for i in range(len(letras)):
	  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)	Eliminar valores de una lista (correcto)

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

En este caso la primera instrucción del bloque comprueba si letras[*i*], es decir, letras[2] es igual a "B". Como letras[2] no existe (porque la lista tiene ahora sólo dos elementos), se produce un error y el programa se interrumpe. La solución es recorrer la lista en orden inverso, los valores que todavía no se han recorrido siguen existiendo en la misma posición que al principio.	

Saber si un valor está o no en una lista

Para saber si un valor está en una lista se puede utilizar el operador in. La sintaxis sería "elemento in lista" y devuelve un valor lógico: True si el elemento está en la lista, False si el elemento **no** está en la listaPor ejemplo, el programa siguiente comprueba si el usuario es una persona autorizada:	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 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 in personas_autorizadas:
  print("Está autorizado")
else:
  print("No está autorizado")
	personas_autorizadas = ["Alberto", "Carmen"]
	nombre = input("Dígame su nombre: ")
	if nombre not in personas_autorizadas:
	  print("No está autorizado")
	else:
	  print("Está autorizado")

El tipo range()

En Python 3, range es un tipo de datos. El tipo range es una **lista inmutable de números enteros en sucesión aritmética**.

- Inmutable significa que, a diferencia de las listas, los range no se pueden modificar.
- Una sucesión artimética es una sucesión en la que la diferencia entre dos términos consecutivos es siempre la misma.

Un range se crea llamando al tipo de datos con uno, dos o tres argumentos numéricos, como si fuera una función. El tipo range() con un único argumento se escribe range(n) y crea una lista inmutable de *n* números enteros consecutivos que empieza en 0 y acaba en *n - 1*.

Para ver los valores del range(), es necesario convertirlo a lista mediante la función list().	El tipo range con dos argumentos se escribe range(m, n) y crea una lista inmutable de enteros consecutivos que empieza en *m* y termina en *n - 1*.	El tipo range con tres argumentos se escribe range(m, n, p) y crea una lista inmutable de enteros que empieza en *m* y acaba justo antes de superar o igualar a *n*, aumentando los valores de *p* en *p*. Si *p* es negativo, los valores van disminuyendo de *p* en *p*.

>>> x = range(10)
>>> x
range(0, 10)
>>> list(x)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> range(7)
range(0, 7)
>>> list(range(7))
[0, 1, 2, 3, 4, 5, 6]
Si *n* no es positivo, se crea un range vacío.
>>> list(range(-2))
[]
>>> list(range(0))
[]
	>>> list(range(5, 10))
	[5, 6, 7, 8, 9]
	>>> list(range(-5, 1))
	[-5, -4, -3, -2, -1, 0]
	Si *n* es menor o igual que *m*, se crea un range vacío.
	>>> list(range(5, 1))
	[]
	>>> list(range(3, 3))
	[]
		>>> list(range(5, 21, 3))
		[5, 8, 11, 14, 17, 20]
		>>> list(range(10, 0, -2))
		[10, 8, 6, 4, 2]
		El valor de *p* no puede ser cero:
		>>> range(4,18,0)
		Traceback (most recent call last):/span>
		File "<pyshell#0>", line 1, in <module>
		range(4,18,0)
		ValueError: range() arg 3 must not be zero
		Si *p* es positivo y *n* menor o igual que *m*, o si *p* es negativo y *n* mayor o igual que *m*, se crea un range vacío.
		>>> list(range(25, 20, 2))
		[]
		>>> list(range(20, 25, -2))
		[]
		En los range(*m*, *n*, *p*), se pueden escribir *p* range distintos que generan el mismo resultado. Por ejemplo:
		>>> list(range(10, 20, 3))
		[10, 13, 16, 19]
		>>> list(range(10, 21, 3))
		[10, 13, 16, 19]
		>>> list(range(10, 22, 3))
		[10, 13, 16, 19]

En resumen, los tres argumentos del tipo range(m, n, p) son:

- *m*: el valor inicial
- *n*: el valor final (que no se alcanza nunca)
- *p*: el paso (la cantidad que se avanza cada vez)

Si se escriben sólo dos argumentos, Python le asigna a *p* el valor 1. Es decir range(m, n) es lo mismo que range(m, n, 1)
Si se escribe sólo un argumento, Python, le asigna a *m* el valor 0 y a *p* el valor 1. Es decir range(n) es lo mismo que range(0, n, 1).

El tipo range() sólo admite argumentos enteros. Si se utilizan argumentos decimales, se produce un error:

```
>>> range(3.5, 10, 2)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    range(3.5, 10, 2)
TypeError: range() integer start argument expected, got float.
```

Concatenar range()

No se pueden concatenar tipos range(), ya que el resultado de la concatenación puede no ser un tipo range()).	Pero sí se pueden concatenar tipos range() previamente convertidos en listas. El resultado es lógicamente una lista, que no se puede convertir a tipo range().	No se pueden concatenar tipos range(), ni aunque el resultado sea una lista de números enteros en sucesión aritmética.

>>> range(3) + range(5)
Traceback (most recent call last):
File "<pyshell#2>", line 1, in <module>
range(3) + range(5)
TypeError: unsupported operand type(s) for +: 'range' and 'range'
	>>> list(range(3)) + list(range(5))
	[0, 1, 2, 0, 1, 2, 3, 4]
		>>> range(1, 3) + range(3, 5)
		Traceback (most recent call last):
		File "<pyshell#3>", line 1, in <module>
		range(3) + range(5)
		TypeError: unsupported operand type(s) for +: 'range' and 'range'
		>>> list(range(1, 3)) + list(range(3, 5))
		[1, 2, 3, 4]

La función len()

La función len() devuelve la longitud de una cadena de caracteres o el número de elementos de una lista.

El argumento de la función len() es la lista o cadena que queremos "medir".	El valor devuelto por la función len() se puede usar como parámetro de range().

>>> len("mensaje secreto")
15
>>> len(["a","b","c"])
3
>>> len(range(1, 100, 7))
15
	>>> list(range(len("mensaje secreto")))
	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
	>>> list(range(len(["a","b","c"])))
	[0, 1, 2]
	>>> list(range(len(range(1, 100, 7))))
	[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]