Sumario

Documentación oficial de Python: https://docs.python.org/3/
Referencia **w3schools**: https://www.w3schools.com/python/python_reference.asp

UT 05: Estructuras de datos

En muchos lenguajes de programación existen estructuras denominadas "arrays" (matrices). El lenguaje Python es mucho más versátil y utiliza estructuras dinámicas, flexibles, que se pueden manejar con gran facilidad, añadir elementos, combinar diferentes tipos, etc... Estas estructuras se denominan **listas** y **tuplas**. Ambas admiten operaciones similares, aunque la diferencia principal entre ambas está en que las tuplas son **inmutables**, es decir, su valor no se puede modificar mediante código (aunque sí se pueden manejar mediante trucos que veremos más adelante).

1 Listas

Las listas son secuencias mutables, formadas por un grupo de valores, separados por comas y encerrados entre corchetes, identificados con un único nombre.

```
numbers = [10, 5, 7, 2, 1]
```

Los elementos dentro de la lista pueden ser de distinto tipo.

```
lista1=[1,"elemento2",3.0]
>>> lista1
[1, 'elemento2', 3.0]
```

Se accede:

- a una lista por su nombre: `nombre_lista`
- a uno de sus elementos con su posición/índice (los ínidices empiezn por 0)
  - elemento i-ésimo: `nombre_lista[indice]`
  - elemento i-ésimo empezando por el final: `nombre_lista[-indice]`

Los elementos de una lista pueden modificarse:

```
numbers = [10, 5, 7, 2, 1]
print("List content:", numbers) # printing original list content
numbers[0]=111
print(numbers)
```

Salida:

```
[111, 5, 7, 2, 1]
```

Se puede utilizar la sentencia "for" para recorrer una lista secuencialmente:

```
for i in secuencia:
     Repite el bucle para cada uno de los elementos de la secuencia
```

No es necesario comprobar la longitud para recorrerla.

Antes de continuar, anticiparemos los conceptos de funciones y métodos:

Tanto **funciones** como **métodos** son conjuntos de instrucciones que realizan una tarea concreta manejando datos. La diferencia estriba en el hecho de que la función maneja datos y puede producir resultados, pero no pertenece a los datos. Por contra, un método es propiedad de un conjunto de datos concreto (más adelante veremos que puede ser un objeto), es decir, realiza operaciones estrictamente sobre los datos a los que pertenece.

- `resultado = función(argumentos)` -> Función
- `resultado = datos.metodo(argumentos)` -> Método

1.1 Funciones de lista: len(lista), del...

`len(lista)` - Devuelve el número de elementos de la lista

`del` …

… `lista[i]` – Elimina el elemento i-ésimo.

… `lista[rodaja]` – Elimina rodajas

… `lista` – Elimina toda la lista

Veamos un ejemplo de uso de estas funciones:

```
numbers = [10, 5, 7, 2, 1]
print("Original list content:", numbers) # printing original list
numbers[0] = 111
print("\nPrevious list content:", numbers) # printing previous list
numbers[1] = numbers[4] # copying fifth element to the second
print("Previous list content:", numbers) # printing previous list
print("\nList's length:", len(numbers)) # printing previous list length

del numbers[1] # removing the second element from the list
print("New list's length:", len(numbers)) # printing new list length
print("\nNew list content:", numbers) # printing current list content
```

Resultados:

```
Original list content: [10, 5, 7, 2, 1]

Previous list content: [111, 5, 7, 2, 1]
Previous list content: [111, 1, 7, 2, 1]
List's length: 5

New list's length: 4
New list content: [111, 7, 2, 1]
```

1.2 Métodos de listas

- `lista.append(elemento)` : introduce elto al final de la lista
- `lista.insert(posicion,elemento)` : introduce elto en esa posición de la lista y todos los elementos a la derecha se trasladan hacia la derecha.
- `lista.sort()` : ordena la lista (si puede)

1.3 Operador "in"

El operador "in" se puede utilizar en conjunción con "for" para recorrer los elementos de una lista o con "if" para devolver un valor lógico:

`... elem in list` - Mira si un elemento está en una lista

`... elem not in list` - Mira si un elemento NO está en una lista

1.4 Listas en memoria

De las listas, se guarda su posición en memoria. En otras palabras, el nombre de la lista es un puntero a una posición de memoria. Cuando se copia una lista en otra, se copian las posiciones de memoria, y por tanto, **la modificación de la lista original afectaría a la lista copiada.**

1.5 Rodajas de listas

Se conocen como "**rodajas**" a conjuntos de elementos pertenecientes a una lista. Se puede así acceder a copias de la lista o de partes de la misma

`lista[inicio:fin]`

Esta expresión sería una copia la lista desde la posición de inicio hasta la posición de (fin-1). Para referirse a una rodaja se pueden usar valores negativos (empezaría por el final). Si no se pone inicio o fin, se asumen principio y final de la lista, respectivamente.

1.6 Listas en listas

Cada elemento de una lista puede ser, por ejemplo, otra lista. De esta forma podemos representar matrices multidimensionales.

Para rellenar los valores de una lista se puede usar un código como este:

```
fila2 = [i**2 for i in range(8)]
>>>fila2
[0, 1, 4, 9, 16, 25, 36, 49]
```

1.7 Ejemplos de operaciones con listas

Veamos algunas operaciones de manejo de listas.

El siguiente código nos permite **invertir una lista**:

```
myList = [10, 1, 8, 3, 5]
length = len(myList)
for i in range(length // 2):
    myList[i], myList[length - i - 1] = myList[length - i - 1], myList[i]
print(myList)
```

Resultado:

```
[5, 3, 8, 1, 10]
```

Ejemplo de **uso de funciones y métodos** para manejar listas: los Beatles...

```
# step 1
beatles=[]
print("Step 1:", beatles)

# step 2
beatles.append("John Lennon")
beatles.append("Paul McCartney")
beatles.append("George Harrison")
print("Step 2:", beatles)

# step 3
for i in ("Stu Sutcliffe","Pete Best"):
    beatles.append(i)
print("Step 3:", beatles)

# step 4
del beatles[3:5]
print("Step 4:", beatles)

# step 5
beatles.insert(0,"Ringo Starr")
print("Step 5:", beatles)

# testing list legth
print("The Fab", len(beatles))
```

Salida:

```
Step 1: []
Step 2: ['John Lennon', 'Paul McCartney', 'George Harrison']
Step 3: ['John Lennon', 'Paul McCartney', 'George Harrison', 'Stu Sutcliffe', 'Pete Best']
Step 4: ['John Lennon', 'Paul McCartney', 'George Harrison']
Step 5: ['Ringo Starr', 'John Lennon', 'Paul McCartney', 'George Harrison']
The Fab 4
```

Resumen del manejo de listas (del curso de Cisco NetAcad):

1. The list is a type of data in Python used to store multiple objects. It is an ordered and mutable collection of comma-separated items between square brackets, e.g.:

```
myList = [1, None, True, "I am a string", 256, 0]
```

2. Lists can be indexed and updated, e.g.:

```
myList = [1, None, True, 'I am a string', 256, 0]
print(myList[3]) # outputs: I am a string
print(myList[-1]) # outputs: 0

myList[1] = '?'
print(myList) # outputs: [1, '?', True, 'I am a string', 256, 0]

myList.insert(0, "first")
myList.append("last")
print(myList) # outputs: ['first', 1, '?', True, 'I am a string', 256,
0, 'last']
```

3. Lists can be nested, e.g.:

```
myList = [1, 'a', ["list", 64, [0, 1], False]].
```

4. List elements and lists can be deleted, e.g.:

```
myList = [1, 2, 3, 4]
del myList[2]
print(myList) # outputs: [1, 2, 4]

del myList # deletes the whole list
```

5. Lists can be iterated through using the for loop, e.g.:

```
myList = ["white", "purple", "blue", "yellow", "green"]

for color in myList:
    print(color)
```

6. The len() function may be used to check the list's length, e.g.:

```
myList = ["white", "purple", "blue", "yellow", "green"]
print(len(myList)) # outputs 5

del myList[2]
print(len(myList)) # outputs 4
```

7. A typical function / method invocation looks as follows:

```
result = function(arg)
result = data.method(arg).
```

Ejercicios con listas:

Exercise 1: What is the output of the following snippet?

```
lst = [1, 2, 3, 4, 5]
lst.insert(1, 6)
del lst[0]
lst.append(1)
print(lst)
```

Salida:
```
[6, 2, 3, 4, 5, 1]
```

Exercise 2: What is the output of the following snippet?

```
lst = [1, 2, 3, 4, 5]
lst2 = []
add = 0
for number in lst:
    add += number
    lst2.append(add)

print(lst2)
```

Salida:
```
[1, 3, 6, 10, 15]
```

Exercise 3: What happens when you run the following snippet?

```
lst = []
del lst
print(lst)
```

Resultado:
```
NameError: name 'lst' is not defined
```

Exercise 4: What is the output of the following snippet?

```
lst = [1, [2, 3], 4]
print(lst[1])
print(len(lst))
```

Resultado:
```
[2, 3]
3
```

1.8 Ordenación de listas: Algoritmo de la burbuja

Cuando tenemos un conjunto de datos, es frecuente que tengamos que realizar operaciones de ordenación de los mismos (de mayor a menor, por orden alfabético, etc...) En los diferentes lenguajes de programación se han desarrollado estrategias para realizar estas operaciones. En muchos casos hay funciones ya programadas en librerías o métodos aplicables a objetos (o conjuntos de objetos) que permiten conseguir la ordenación de los datos, con diferentes niveles de eficiencia. Los algoritmos de ordenación suelen ser más sencillos y fáciles de entender, pero más ineficientes en el sentido de que requieren más tiempo o potencia de proceso, porque requieren un mayor número de operaciones de cálculo y/o acceso a memoria. Por contra, los métodos de ordenación más eficientes son complejos y, en ocasiones, difíciles de entender.

No vamos a detallar en este capítulo los diferentes algoritmos y sus características, pero sí vamos a presentar al menos alguna estrategia que se puede conseguir para lograr el objetivo. Típicamente se suele hacer referencia al ***"algoritmo de la burbuja"***:

Si partimos de un conjunto de datos desordenados, podemos conseguir ordenarlos (pro ejemplo, de menor a mayor) realizando una serie de pasos:

1. Recorremos todos los elementos de izquierda a derecha, comparando cada uno de ellos con su elemento contiguo.
2. Si el primer elemento es mayor, se intercambian las posiciones.
3. La operación se repite hasta que se llega al último elemento. Al final, el último elemento será el mayor de la lista.
4. El proceso comienza de nuevo con el conjunto de datos restante (excepto el último elemento, que ya está ordenado) y así sucesivamente hasta que queden todos ordenados (si hay n elementos, se realizarán n-1 iteraciones)

Ejemplo: veamos cómo ordenar una lista de 5 elementos en 4 iteraciones.

Dada la lista original:

9	10	5	2	3

- Primera iteración: 4 pasos

Como 9 < 10, en el primer paso no se cambia nada

9	10	5	2	3
9	5	10	2	3
9	5	2	10	3
9	5	2	3	10

Tras recorrer todos los elementos por primera vez de izquierda a derecha, el elemento 10 (el mayor de la lista) ya ha alcanzado la última posición.

- En la segunda iteración realizamos 3 pasos. Comenzamos con esta situación:

9	5	2	3	10

Recorremos los elementos hasta el penúltimo:

5 →	9	2	3	10
5	2 →	9	3	10
5	2	3 →	9	10

- En la tercera iteración nos bastarán dos pasos. Partiendo de esta situación inicial:

5	2	3	9	10

Recorremos los elementos salvo los dos últimos

2 →	5	3	9	10
2	3 →	5	9	10

Llegados a este punto tenemos ordenados los tres últimos elementos.

- En la cuarta y útlima iteracion solo faltan por ordenar los dos primeros elementos. El algoritmo realiza la comparación, pero en este caso no necesita realizar ningún intercambio y este sería el resultado final

2	3	5	9	10

Veamos el código que implementa este algoritmo en Python:

```
myList = [9, 10, 5, 2, 3] # list to sort
for j in range(1,len(myList)-1):
    for i in range(len(myList) - j): # we need (5 - 1) comparisons
        if myList[i] > myList[i + 1]: # compare adjacent elements
            myList[i], myList[i + 1] = myList[i + 1], myList[i] # swap
    print(myList)
```

Resultado:

```
[9, 5, 2, 3, 10]
[5, 2, 3, 9, 10]
[2, 3, 5, 9, 10]
```

NOTA: Para **intercambiar valores de variables** no necesitamos emplear una variable auxiliar. Basta con una expresión como esta:

```
var1, var2 = var2, var1
```

Otra forma más eficiente de escribir el código sería:

```
myList = [9, 10, 5, 2, 3] # list to sort
swapped = True # it's a little fake - we need it to enter the while loop
while swapped:
    swapped = False # no swaps so far
    for i in range(len(myList) - 1):
        if myList[i] > myList[i + 1]:
            swapped = True # swap occured!
            myList[i], myList[i + 1] = myList[i + 1], myList[i]
```

En el caso de Python, ni siquiera es necesario recurrir a estos algoritmos, porque las listas ya tienen algunos métodos implementados, como "sort". Nuestro programa quedaría reducido a:

```
myList = [8, 10, 6, 2, 4]
myList.sort()
print(myList)
```

El método "sort" es muy potente y permite ordenar diferentes tipos de datos de forma inteligente.

```
lst = ["D", "F", "A", "Z"]
lst.sort()
print(lst)
```

Salida:

```
['A', 'D', 'F', 'Z']
```

Podemos utilizar otros métodos, como "reverse":

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

Salida:

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

Ejercicios resueltos:

1) Obtener el mayor valor de una lista

```
myList = [17, 3, 11, 5, 1, 9, 7, 15, 13]
largest = myList[0]
for i in range(1, len(myList)):
    if myList[i] > largest:
        largest = myList[i]

print(largest)
```

2) Encontrar un valor en una lista:

```
myList = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
toFind = 5
found = False

for i in range(len(myList)):
    found = myList[i] == toFind
    if found:
        break

if found:
    print("Element found at index", i)
else:
    print("absent")
```

3) La lotería primitiva.

```
drawn = [5, 11, 9, 42, 3, 49]
bets = [3, 7, 11, 42, 34, 49]
hits = 0

for number in bets:
    if number in drawn:
        hits += 1

print(hits,"aciertos")
```

4) Programa para eliminar números repetidos en una lista:

```
myList = [1, 2, 4, 4, 1, 4, 2, 6, 2, 9]
#
# put your code here
#
list2=myList[:]
for i in myList:
    if (i not in list2):
        list2.append(i)
    else:
        del(myList[i])
print("The list with unique elements only:")
print(myList)
```

1.9 Matrices (listas en listas) y sintaxis "list comprehension"

Python nos permite expresar asignaciones a listas con una curiosa sintaxis:

```
squares = [x ** 2 for x in range(10)]
```

"squares" sería una lista formada por 10 elementos, que corresponden con los 10 primeros números naturales al cuadrado (del 0 al 9).

O por ejemplo, si queremos que considere solo los números impares, podemos escribir:

```
odds = [x for x in squares if x%2!=0]
```

Un tablero de ajedrez podría ser una **matriz** vacía, como esta:

```
EMPTY=""
board = []

for i in range(8):
    row = [EMPTY for i in range(8)]
    board.append(row)
print (board)
```

Producirá esta salida:

```
[['', '', '', '', '', '', '', ''],
(...8 filas...)
['', '', '', '', '', '', '', '']]
```

O bien, una forma aún más corta para crear el tablero, usando "**list comprehension**":

```
EMPTY=""
board = [[EMPTY for i in range(8)] for j in range(8)]
```

En ocasiones llenamos matrices con valores aleatorios (solo para pruebas). Por ejemplo, veamos cómo definir una matriz de temperaturas máximas a lo largo de un mes

```
from random import *
hitemps=[[int(random()*20+15)  for  hour  in  range(24)]  for  month  in
range(8)]
print(hitemps)
```

En resumen, la sintaxis **list comprehension** de Python nos permite hacer esto:

- Opción 1:

```
[expression for element in list if conditional]
```

- Opción 2

```
for element in list:
    if conditional:
        expression
```

2 Tuplas

Hemos visto que Python ofrece la posibilidad de trabajar con **secuencias**, tipos de datos que permiten almacenar más de un valor y pueden ser iterados (recorridos secuencialmente) mediante un bucle "`for`". Un ejemplo son las listas.

La **mutabilidad** es una propiedad de los datos. Indica la posibilidad de que un dato determinado pueda ser modificado o no durante la ejecución de un programa. Existen tipos de datos mutables e inmutables. Los tipos de datos mutables pueden ser actualizados en cualquier momento. **Los datos inmutables no pueden ser modificados**. Las listas son mutables. Por ejemplo, se pueden añadir elementos con `append()`.

Existe otro tipo de secuencias, las **tuplas**, que son **inmutables**. También están compuestas por un conjunto de datos iterables, pero no pueden ser modificadas (no se puede alterar el valor de los datos almacenados ni existen métodos para añadir o borrar).

Las tuplas se suelen definir entre paréntesis o expresando la lista de elementos separados por comas. Para crear una tupla de un solo elemento hay que usar la coma.

```
tupla1 = (1,2,3)
tupla2 = 3.4, 2., 5.0
```

A efectos de manipulación se comportan como listas. Se usan como listas, incluidas funciones, métodos, operadores, rodajas… excepto aquellos métodos que suponen la modificación de su contenido (append, del...). Ejemplo:

```
miTupla = (1, 10, 100, 1000)
print(miTupla[0], end=" - ")
print(miTupla[-1], end=" - ")
print(miTupla[1:], end=" - ")
print(miTupla[:-2])
```

Resultado:

```
1 - 1000 - (10, 100, 1000) - (1, 10)
```

Como los datos pueden modificarse, para cambiar el valor de los elementos de una tupla se utilizan estrategias como la concatenación, asignación múltiple…

```
var = 123
t1 = (1, )
t2 = (2, )
t3 = (3, var)
t1, t2, t3 = t2, t3, t1
print(t1, t2, t3)
```

Resultado:

```
(2,) (3, 123) (1,)
```

3 Diccionarios

Los diccionarios son estructuras de datos diferentes. **No son secuencias** y **son mutables**.

Un **diccionario** es un conjunto de pares **clave-valor**.

Consideraciones:

- Cada clave debe ser única.
- La clave puede ser de cualquier tipo de datos
- No son listas
- La función `len()` funciona
- Un diccionario es NO ordenado
- Las relaciones clave-valor no son unívocas. Solo funcionan en un sentido, es decir, a cada clave única le corresponde un único valor, pero varias claves pueden tener el mismo valor.

Métodos en diccionarios:

- `keys()` : devuelven la lista de las claves
- `values()` : devuelve la lista de los valores
- `items()` : devuelve una lista de tuplas, con los pares clave-valor.
- `k = nombrediccionario [clave]` : devuelve una tupla con el valor.
- Para acceder al valor tal cual : `k[0]`

Insertando y borrando en diccionarios:

- Para insertar, se asigna un nuevo valor y ya está.
- Para cambiar un valor, se le asigna el nuevo valor y ya está.
- Para borrar, se usa la función `del`

Ejemplo de creación de diccionario y acceso a elementos:

```
dict = {"cat" : "chat", "dog" : "chien", "horse" : "cheval"}
phoneNumbers = {'boss' : 5551234567, 'Suzy' : 22657854310}
emptyDictionary = {}
print(dict)
print(phoneNumbers)
print(emptyDictionary)
print(dict['dog'])
print(phoneNumbers['Suzy'])
for key in dict.keys():
    print(key, "->", dict[key])
```

Resultado:

```
{'cat': 'chat', 'dog': 'chien', 'horse': 'cheval'}
{'boss': 5551234567, 'Suzy': 22657854310}
{}
chien
22657854310
cat -> chat
dog -> chien
horse -> cheval
```

También se puede utilizar la función "`sorted()`", para obtener una lista ordenada (en la medida de lo posible, porque hay datos que no se pueden ordenar). Para añadir una nueva clave basta con invocar al diccionario y realizar una asignación. También se puede añadir un elemento con el método "`update`" y eliminar una clave con la sentencia "`del`":

```
dict = {"door" : "puerta", "class" : "clase", "Student" : "alumno"}

for english, spanish in sorted(dict.items()):
    print(english, "->", spanish)
# Modificar un valor:
dict['Student']='Estudiante'
print(dict['Student'])
# Añadir un nuevo valor:
dict['building']='edificio'
dict.update({'room':'sala'})
del dict['class']
print(dict)
```

Salida:

```
Student -> alumno
class -> clase
door -> puerta
Estudiante
{'door': 'puerta', 'Student': 'Estudiante', 'building': 'edificio', 'room':
'sala'}
```

4 Utilizando conjuntamente tuplas y diccionarios

Es posible utilizar conjuntamente tuplas y diccionarios. A continuación veremos un ejemplo, en el que diseñaremos un programa para almacenar las notas de los alumnos de una clase.

El programa irá preguntando sucesivamente un nombre de alumno y una nota, hasta que el usuario introduzca la secuencia "exit". Lógicamente, es posible repetir el nombre del

alumno, asignándole un número indefinido de notas. Cuando se teclee "exit", el programa presentará las notas medias obtenidas por todos los alumnos existentes.

Cabe destacar que en este ejemplo usamos tuplas, por definición inmutables, pero cada vez que el usuario introduce la nota de un alumno, esta se añade a la tupla de notas. En realidad, no estamos modificándola, sino que creamos una nueva tupla con el mismo nombre, que se forma concatenando el contenido anterior con el nuevo valor añadido.

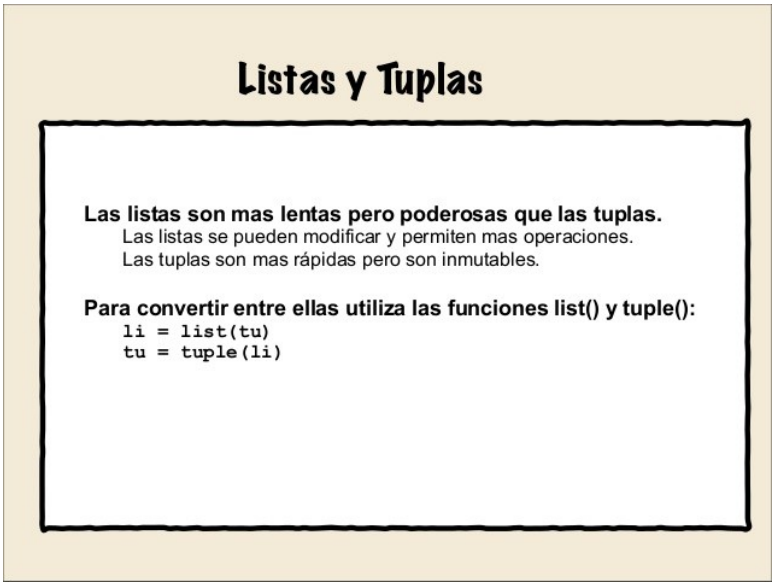
Este es el código:

```
schoolClass = {}
while True:
    name = input("Enter the student's name (or type exit to stop): ")
    if name == 'exit':
        break

    score = int(input("Enter the student's score (0-10): "))

    if name in schoolClass:
        schoolClass[name] += (score,)
    else:
        schoolClass[name] = (score,)

for name in sorted(schoolClass.keys()):
    sum = 0
    counter = 0
    for score in schoolClass[name]:
        sum += score
        counter += 1
    print(name, ":", sum / counter)
```



5 Cadenas de caracteres

Los "Strings" son secuencias inmutables de caracteres. Se manejan con funciones propias de secuencias:

len()

```
subcadena in cadena
```

```
for i in cadena:
```

`cadena[i]` => para acceder al carácter i+1-ésimo
`cadena[i:j]` => para acceder a un substring

5.1 Codificación de caracteres: ASCII, Unicode, UTF-8

En realidad, Los caracteres se guardan como valores enteros, en codificación UTF-8. La función `ord()` devuelve el código numérico asociado a un carácter, y la función `chr()` devuelve el carácter correspondiente a un código numérico.

La codificación de caracteres más simple y antigua es ASCII. Aquí tenemos la tabla de códigos de 8 bits (256 valores, de los que se usan 128):

Character	Code	Character	Code	Character	Code	Character	Code
(NUL)	0	(space)	32	@	64	`	96
(SOH)	1	!	33	A	65	a	97
(STX)	2	"	34	B	66	b	98
(ETX)	3	#	35	C	67	c	99
(EOT)	4	$	36	D	68	d	100
(ENQ)	5	%	37	E	69	e	101
(ACK)	6	&	38	F	70	f	102
(BEL)	7	'	39	G	71	g	103
(BS)	8	(	40	H	72	h	104
(HT)	9	)	41	I	73	i	105
(LF)	10	*	42	J	74	j	106
(VT)	11	+	43	K	75	k	107
(FF)	12	,	44	L	76	l	108
(CR)	13	-	45	M	77	m	109
(SO)	14	.	46	N	78	n	110
(SI)	15	/	47	O	79	o	111

(DLE)	16	0	48	P	80	p	112
(DC1)	17	1	49	Q	81	q	113
(DC2)	18	2	50	R	82	r	114
(DC3)	19	3	51	S	83	s	115
(DC4)	20	4	52	T	84	t	116
(NAK)	21	5	53	U	85	u	117
(SYN)	22	6	54	V	86	v	118
(ETB)	23	7	55	W	87	w	119
(CAN)	24	8	56	X	88	x	120
(EM)	25	9	57	Y	89	y	121
(SUB)	26	:	58	Z	90	z	122
(ESC)	27	;	59	[	91	{	123
(FS)	28	<	60	\	92	|	124
(GS)	29	=	61	]	93	}	125
(RS)	30	>	62	^	94	~	126
(US)	31	?	63	_	95		127

Esta codificación resultaba claramente insuficiente para represenar los alfabetos internacionales. Un primera aproximación para resolver el problema consistió en diseñar diferentes páginas de códigos en función de los lenguajes utilizados. Surgieron diferentes formatos que no prosperaron hasta que apareció el estándar **Unicode**, que asigna un único carácter a cada código. Los primeros 128 elementos coinciden con ASCII, y a su vez, los primeros 256 son idénticos a los representados por la página de códigos ISO/IEC 8859-1 (idiomas europeos occidentales). Hay varias implementaciones de Unicode, como **UCS (Unicode Standard),** basada en 32 bits, o **UTF-8 (Unicode Transformation Format)**, que es la más usada, y se caracteriza por representar los caracteres con tantos bits como sea realmente necesario. Por ejemplo, utiliza 8 bits para los caracteres latinos, 16 bits para los caracteres no latinos y 24 bits para las ideografías propias de los idiomas chino, japonés y coreano. Python usa codificación **UTF-8**.

5.2 Funciones para strings

- `min(cadena)` => Devuelve el carácter “menor” de los incluidos en cadena (aquel cuyo código numérico es menor)
- `max(cadena)` => Devuelve el carácter “mayor” de los incluidos en cadena (el de mayor código numérico asociado)
- `list(cadena)` => Devuelve una lista con los caracteres de la cadena

5.3 Métodos para strings – encontrar y contar

- `index(subcadena)` => Devuelve la posición de la primera ocurrencia de la subcadena en la cadena. La subcadena debe estar en la cadena. En caso contrario genera una excepción ValueError.
- `find(subcadena)` => Devuelve la posición de la primera ocurrencia de la subcadena en la cadena, o "-1" Si no está. Si se da un segundo argumento, empieza a buscar desde esa posición. Con un tercer argumento, acaba de buscar en tal posición.
- `rfind(subcadena)` => Hace lo mismo que find, empezando por la derecha.
- `count(subcadena)` => Devuelve el número de veces que aparece la subcadena.
- `capitalize()` => Devuelve una nueva cadena con la inicial en mayúsculas.
- `lower()` => Pasa a minúsculas.
- `upper()` => Pasa a mayúsculas.
- `endswith(subcadena)` => Devuelve un valor booleano que nos dice si la cadena acaba en esa subcadena.
- `startswith(subcadena)` => Devuelve un valor booleano que nos dice si la cadena empieza con esa subcadena.

5.4 Métodos para strings - contenido

- `isalnum()` => Devuelve un booleano indicando si el string contiene sólo caracteres alfanuméricos (letras y dígitos)
- `isalpha()` => Devuelve `true` si el string contiene sólo letras
- `isdigit()` => Devuelve `true` si el string contiene sólo números
- `islower()` => Devuelve `true` si el string contiene sólo letras minúsculas
- `isupper()` => Devuelve `true` si el string contiene sólo letras mayúsculas
- `isspace()` => Devuelve un booleano indicando si el string contiene sólo espacios en blanco (incluye los saltos de línea)

5.5 Otros métodos para strings - manipular cadenas

- `lstrip(caracteres)` => a la izquierda (si no tiene argumentos quita los blancos)
- `rstrip(caracteres)` => a la derecha
- `strip(caracteres)` => a la izquierda y a la derecha
- `replace(subcadena1,subcadena2)` => Reemplaza la primera por la segunda subcadena. Si se le da un tercer argumento (un entero) se le indica el número máximo de remplazamientos
- `join(lista)` => Genera un string donde se unen los elementos de la lista. Lo que les une es el string desde el que se invoca el método.
- `split()` => Genera una lista con substrings (delimitados por espacios en blanco)

5.6 Comparando strings

Se usan los comparadores habituales, teniendo en cuenta que se comparan los códigos numéricos que representan los caracteres. Es sensible a mayúsculas y minúsculas

`==` => tienen que ser exactamente iguales

`<` => se compara el primer carácter distinto

Si son iguales, pero uno de los strings es más largo, este se considera mayor.

5.7 Ordenando listas (y strings)

- Función `sorted(lista)` => Devuelve una nueva lista ordenada
- Método `sort()` => Ordena la lista in situ

5.8 Ejemplos de métodos y funciones con strings

```
>>> print("aAbByYzZaA".index("b"))
2

>>> print("aAbByYzZaA".index("Z"))
7

>>> print('abcabc'.count("d"))
0

>>> print("abcabc".count("b"))
2

>>> print('aBcD'.capitalize())
Abcd

>>> print('[' + 'alpha'.center(10) + ']')
[  alpha   ]

>>> if "epsilon".endswith("on"):
...     print("yes")
... else:
...     print("no")
...
yes

>>> print("Eta".find("ta"))
1

>>> print('lambda30'.isalnum())
True

>>> print('lambda'.isalnum())
True
```

```
>>> print('30'.isalnum())
True

>>> print('@'.isalnum())
False

>>> print('lambda_30'.isalnum())
False

>>> print("Moooo".isalpha())
True

>>> print('Mu40'.isalpha())
False

>>> print('2018'.isdigit())
True

>>> print("Year2019".isdigit())
False

>>> print(",".join(["omicron", "pi", "rho"]))
omicron,pi,rho

>>> print("SiGmA=60".lower())
sigma=60

>>> print("[" + " tau ".lstrip() + "]")
[tau ]

>>> print("www.cisco.com".lstrip("w."))
cisco.com

>>> print("This is it!".replace("is", "are"))
Thare are it!

>>> print("tau tau tau".rfind("ta"))
8

>>> print("tau tau tau".rfind("ta", 3, 9))
4

>>> print("[" + " upsilon ".rstrip() + "]")
[ upsilon]

>>> print("cisco.com".rstrip(".com"))
cis

>>> print("phi       chi\npsi".split())
['phi', 'chi', 'psi']

>>> print("omega".startswith("meg"))
False
```

```
>>> print("[" + "   aleph   ".strip() + "]")
[aleph]

>>> print("I know that I know nothing. Part 1.".title())
I Know That I Know Nothing. Part 1.
```

5.9 Ejercicio: Marcador LED numérico

Diseñar un display electrónico de 7 segmentos. Los números se representarán mediante matrices de 3x5 en las que cada posición es un LED que puede estar encendido (representado por el carácter "#") o apagado (espacio " "). Ejemplo:

```
  # ### ### # # ### ### ### ### ### ###
  #   #   # # # #   #     # # # # # # #
  # ### ### ### ### ###   # ### ### # #
  # #     #   #   # # #   # # #   # # #
  # ### ###   # ### ###   # ### ### ###
```

El programa permitirá representar cifras repetidamente hasta que el usuario introduzca un valor negativo.

Solución:

```
numeros={'0':['###','# #','# #','# #','###'],
        '1':[' # ',' # ',' # ',' # ',' # '],
        '2':['###','  #','###','#  ','###'],
        '3':['###','  #','###','  #','###'],
        '4':['# #','# #','###','  #','  #'],
        '5':['###','#  ','###','  #','###'],
        '6':['###','#  ','###','# #','###'],
        '7':['###','  #','  #','  #','  #'],
        '8':['###','# #','###','# #','###'],
        '9':['###','# #','###','  #','###']}

numero=0

while numero>=0:
    try:
        numero=int(input("Introduzca un número entero (<0 para finalizar): "))
    except:
        print("Valor introducido no válido")
    else:
        try:
            for i in range(5):
                linea=""
                for digito in str(numero):
                    linea=linea+numeros[digito][i]+" "
                print(linea)
        except KeyError:
            print("Hasta pronto!")
        except:
            print("Error desconocido")
```