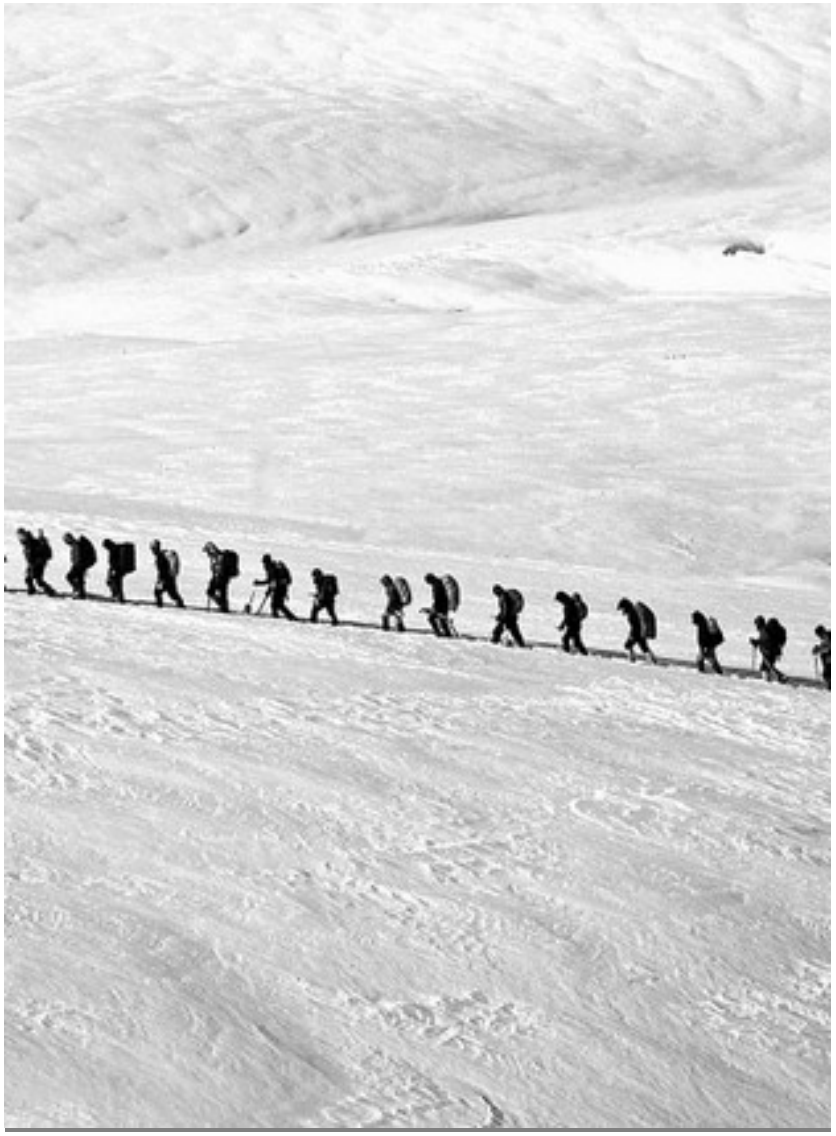
Fundamentos de Programación. Primer Curso de ASIR.

UT04.01 - Estructuras de datos. Listas.

UT04.01 - Estructuras de datos. Listas.
1.- Introducción.
1.1.- Estructuras de datos de datos.

- En muchos lenguajes de programación existen estructuras denominadas **arrays**.
- En Python hay estructuras más versátiles que permite manejar **grupos de elementos**, realizando multitud de operaciones: añadir, eliminar, buscar, etc.
- Estas estructuras son las siguientes: l**istas, tuplas y diccionarios**.
- Todas ellas admiten operaciones similares, aunque la diferencia principal está en que se pueda cambiar sus elementos (**mutables**) o no (**inmutables**).
- En este documento trataremos la primera de ellas: **las listas**.



UT04.01 - Estructuras de datos. Listas.
1.- Introducción.
1.2.- Definición de lista. Características (I).

Definición.

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

  numeros = [10, 5, 7, 2, 1]

- Los elementos dentro de la lista pueden ser de distinto tipo de datos. Es decir, pueden ser números, cadenas e incluso otras estructuras.

  lista = [1, ”elemento2”, 3.175]

Acceso a una lista y sus elementos.

- A una lista se accede por su nombre.

  print(lista)

- A los elementos de una lista se accede por el nombre de la lista y índice, **empezando por el cero**. Se puede acceder de dos formas:
  - **Desde el principio**: **lista[indice]**.
  - **Desde el final**: **lista[-indice]**. Empieza en menos uno.

```
>>> numeros = [3,5,7,4,8,10]
>>> numeros[4]
8
>>> numeros[-2]
8
>>> numeros[-4]
7
>>> print(numeros)
[3, 5, 7, 4, 8, 10]
```

UT04.01 - Estructuras de datos. Listas.
1.- Introducción.
1.2.- Definición de lista. Características (II).

- Cuando se crea una lista, la variable lista apunta a una dirección de memoria en donde están almacenados secuencialmente los distintos elementos.
- En programación se dice que la lista **es un puntero a una posición de memoria**.
- Cuando se copia una lista en otra, se copian las posiciones de memoria. Por tanto, **la modificación de la lista original afectaría a la lista copiada**.
- En el ejemplo vemos que la lista b se iguala a la lista a. Al modificar a automáticamente se modifica b.

```
>>> a = [1,3,4]
>>> b = a
>>> a[1] = 100
>>> b
[1, 100, 4]
```

UT04.01 - Estructuras de datos. Listas.
2.- Modificación y recorrido de una lista.

Modificación de los elementos de una lista.

- Los elementos de una lista pueden modificarse. Simplemente hay que asignar el valor a la posición deseada.

  numeros[2] = 15

Recorrido de una lista.

- Es posible recorrer una lista mediante un bucle for in. La sintaxis es la siguiente:

  for elemento in lista:
  [Secuencia de comandos]

```
>>> numeros = [2,4,1,7,3,10]
for numero in numeros:
    print(numero)

2
4
1
7
3
10

>>> numeros[3] = 100
>>> numeros
[2, 4, 1, 100, 3, 10]
```

UT04.01 - Estructuras de datos. Listas.
3.- Creación rápida de listas. List Comprehension (I).

- Las comprensiones de listas permiten crear listas de forma más concisa. Pueden ser útiles cuando quieres crear nuevas listas a partir de listas existentes o iterables.
- Una forma simple puede ser la siguiente:

  [expresión for elemento in iterable]

  En el primer ejemplo se crea una lista con el cuadrado de los cuatro primeros números.
- Es posible también usar un condicional if para filtrar ciertos valores de la lista final.

  [expression for elemento in iterable if condition]
- En el segundo ejemplo se consiguen todos los números pares en el rango dado.

```
>>> numeros = [num**2 for num in range(1,5)]
>>> numeros
[1, 4, 9, 16]
>>> numeros
[1, 4, 9, 16]

>>> numeros = [num for num in range(1,10)
... if num % 2 ==0]
>>> numeros
[2, 4, 6, 8]
```

UT04.01 - Estructuras de datos. Listas.
3.- Creación rápida de listas. List Comprehension (II).

- Un ejemplo mas complejo de usar comprensiones de lista sería agregando expresiones condicionales if-else dentro de ellas.
- Las posiciones para el bucle for y el condicional son intercambiadas.

  La sintaxis es:

  [expression1 if condición
  else expresion2
  for elemento in iterable]
- En ejemplo siguiente se muestra True si el elemento es impar y False si el elemento es par.
- Un caso mas sofisticado es cuando tenemos varios índices asociados a varios for. En este caso se hacen combinaciones de elementos que cumplen esa condición.

  Véase el último de los ejemplos.

```
>>> lista = [True if num % 2 ==1 else False
... for num in range(1,5)]
>>> lista
[True, False, True, False]

>>> [(x, y) for x in [1,2,3]
... for y in [3,1,4] if x != y]
[(1,3),(1,4),(2,3),(2,1),(2,4),(3,1),(3,4)]
```

UT04.01 - Estructuras de datos. Listas.
4.- Sublistas de una lista (Rebanadas).

- Es posible seleccionar partes de una lista utilizando **rangos de valores**. A esta técnica se la denomina **hacer una rebanada de una lista**.
  - **[n:m:s]**. Empieza en la posición n hasta la anterior a la posición m dando saltos de s elementos.
  - **[n:]**. Desde la posición n hasta el final.
  - **[:n]**. Desde el principio hasta la posición n.
- Es útil recordar que:
  - Es posible empezar a contar desde el final de la lista usando números negativos.
  - Si se especifica un salto negativo la lista se recorre desde el final.

```
>>> numeros
[2, 4, 1, 100, 3, 10]
>>> numeros[1:5]
[4, 1, 100, 3]
>>> numeros[3:]
[100, 3, 10]
>>> numeros[0::2]
[2, 1, 3]
>>> numeros[::-1]
[10, 3, 100, 1, 4, 2]
```

UT04.01 - Estructuras de datos. Listas.
5.- Preguntar si un elemento está o no en una lista.

- Podemos preguntar si un determinado elemento está en una lista mediante el operador **in** o **not in**.
- El operador **in** funciona de la siguiente manera:
  - True si el elemento está en la lista.
  - False si el elemento está en la lista.
- Análogamente, el operador **not in** funciona de la siguiente manera:
  - True si el elemento está en la lista.
  - False si el elemento está en la lista.

```
>>> colores = ["Rojo", "Verde", "Azul"]
>>> "Verde" in colores
True
>>> "Amarillo" in colores
False
>>> "Amarillo" not in colores
True
>>> "Azul" not in colores
False
```

UT04.01 - Estructuras de datos. Listas.
6.- Listas de listas (Matrices).
6.1.- Acceso a una fila y un elemento en particular.

- Una lista puede tener cualquier tipo de elementos, **incluidas otras listas**.
- En estos casos podemos referirnos a elemento de la lista principal (**sublista**) mediante su indice en la lista principal.

  sublista = listas[n]

- Otra posibilidad es que tengamos que referirnos a un elemento de una sublista en particular. En este caso, usaremos dos índices: uno para el elemento sublista y otro el elemento de la sublista.

  elemento = listas[n][m]

```
>>> listas =[[1,4,5],[2,1],[7,2]]
>>> listas[1]
[2, 1]
>>> listas[2][0]
7

>>> for sublista in listas:
...     for elemento in sublista:
...             if (elemento>=5):
...                     print(f"{elemento} esta en la lista {sublista}.")
...
5 esta en la lista [1, 4, 5].
7 esta en la lista [7, 2].
```

UT04.01 - Estructuras de datos. Listas.
6.- Listas de listas (Matrices).
6.2.- Acceso a las columnas de una matriz.

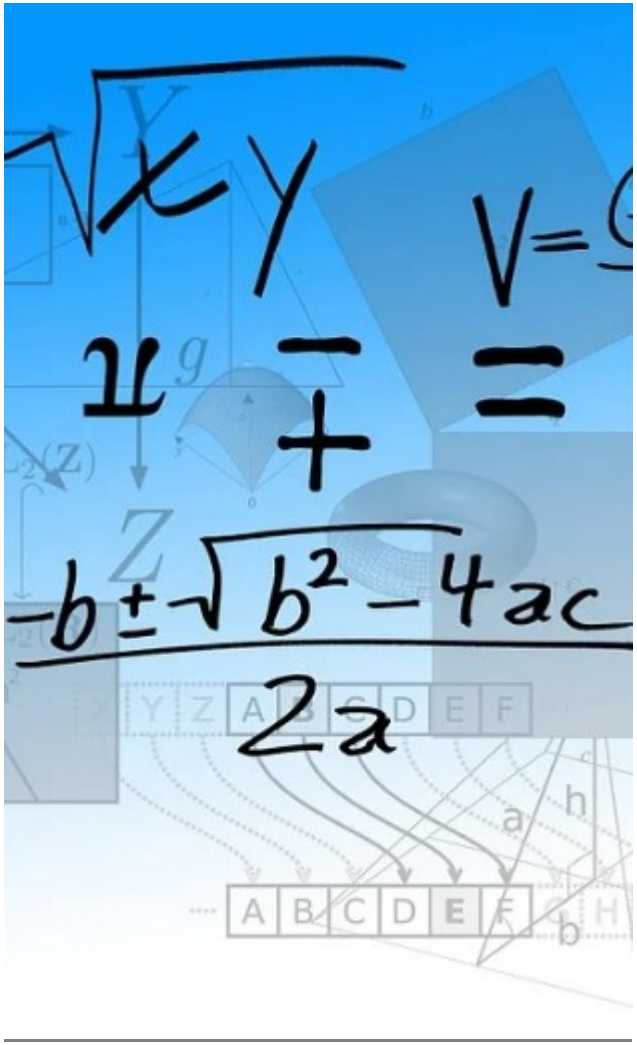
- Es posible acceder a todos los elemento de índice n de una matriz de listas.
- Para ello se especifica mediante la sintaxis

  row[n]

- Veamos un ejemplo.

```
>>> matriz = [[1,4,5], [0,3,2], [7,-2,1]]
>>> [row[0] for row in matriz]
[1, 0, 7]
>>> transpuesta = [ [row[i] for row in matriz] for i in range(3) ]
>>> transpuesta
[[1, 0, 7], [4, 3, -2], [5, 2, 1]]
```

UT04.01 - Estructuras de datos. Listas.
7.- Funciones y métodos de listas.



- 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 función maneja datos y puede producir resultados, pero no pertenece a los datos.
- Un método es propiedad de un objeto y realiza operaciones estrictamente sobre los datos a los que pertenece.
- Para llamar una función usaremos la sintaxis:

  resultado = función(argumentos)
- Mientras que para llamar a un método usaremos la sintaxis:

  resultado = datos.metodo(argumentos)

UT04.01 - Estructuras de datos. Listas.
7.- Funciones y métodos de listas.
7.1.- Funciones que operan sobre listas.

- En la versión anterior de Python (Python2) había multitud de funciones para manipular listas.
- En Python3 tan solo se mantiene la función **len**. La función len devuelve la longitud de una lista; es decir el número de elementos que contiene.
- Su sintaxis es la siguiente:

  len(lista)

```
>>> lista = ["Rojo", "Verde", "Azul", "Amarillo", "Verde"]
>>> len(lista)
5

>>> for indice in range(len(lista)):
...     print(f"lista[{indice}] = {lista[indice]}")
...
lista[0] = Rojo
lista[1] = Verde
lista[2] = Azul
lista[3] = Amarillo
lista[4] = Verde
```

UT04.01 - Estructuras de datos. Listas.
7.- Funciones y métodos de listas.
7.2.- Métodos de listas (I).

lista = [1, 3, 5, 7]			
Método	**Función**	**Ejemplo**	**Resultado**
lista.append()	Añade un nuevo elemento **al final de la lista**.	lista.append(10)	[1, 3, 5, 7, 10]
lista.extend()	Añade un grupo de elementos (iterables) al final de la lista.	lista.extend([2, 5])	[1, 3, 5, 7, 2, 5]
lista.insert(n,elemento)	Inserta un elemento **en una posición de índice n de la lista**.	lista.insert(2,4)	[1, 3, 4, 5, 7]
lista.remove(elemento)	Elimina la primera ocurrencia del elemento en la lista.	lista.remove(5)	[1, 3, 7]
lista.pop(indice)	Obtiene y elimina el elemento de la lista en la posición n. Si no se especifica, obtiene y elimina el último elemento.	lista.pop(1) lista.pop() lista	3 1 [7]
lista.index(elemento)	Obtiene el índice de la primera ocurrencia del elemento en la lista. Si el elemento no se encuentra, se lanza la excepción ValueError. Se pueden añadir indices de inicio y final para buscar en rebanadas de la lista.	lista.append(3) lista.index(3)	[1, 3, 5, 7, 3] 1

UT04.01 - Estructuras de datos. Listas.
7.- Funciones y métodos de listas.
7.2.- Métodos de listas (II).

lista = [3,2,5,3]			
Método	**Función**	**Ejemplo**	**Resultado**
lista.count(elemento)	Devuelve el número de ocurrencias del elemento en la lista.	lista.count(3)	2
lista.reverse()	Obtiene los elementos de la lista en orden inverso.	lista.reverse()	[3,5,2,3]
lista.sort()	Ordena la lista en función de la función menor. Para otros tipos de ordenaciones revisar la documentación de Python.	lista.sort() lista.sort(reverse=True)	[2,3,3,5] [5,3,3,2]
lista.copy()	Retorna una copia superficial de la lista. Equivalente a lista[:].	otraLista = lista.copy() otraLista	 [3,2,5,3]
lista.clear()	Borra todos los elementos de la lista.	lista.clear()	[]

- Para una información mas detallada sobre Estructuras de datos de datos revisar la documentación oficial de Python en la dirección.

https://docs.python.org/es/3/tutorial/datastructures.html

UT04.01 - Estructuras de datos. Listas.
7.- Funciones y métodos de listas.
7.3.- La instrucción del.

- La instrucción del sirve como norma general para eliminar una variable del entorno. De esta forma se libera la memoria que usa. El uso de esta instrucción es la siguiente:

  del variable

- En el caso de las listas nos va servir para dos cosas:
  - Eliminar un elemento de la lista mediante el comando **del lista[indice]**.
  - Eliminar la lista entera. Aquí se actúa como una variable normal con el comando **del lista**.

```
>>> colores = ["Verde", "Amarillo", "Blanco", "Rojo"]
>>> del colores[2]
>>> colores
['Verde', 'Amarillo', 'Rojo']
>>> del colores
>>> colores
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'colores' is not defined
```

UT04.01 - Estructuras de datos. Listas.
8.- Listas y cadenas.
8.1.- Obtener una lista a partir de una cadena. La función list().

- La función **list()** convierte la cadena en una lista en Python.
- La función list() toma una cadena como argumento y la convierte en una lista. Cada carácter de la cadena se convierte en el elemento individual de la lista.
- La sintaxis es la siguiente:

  lista = list(cadena)

```
>>> cadena = "Hola Mundo"
>>> lista = list(cadena)
>>> lista
['H', 'o', 'l', 'a', ' ', 'M', 'u', 'n', 'd', 'o']
```

UT04.01 - Estructuras de datos. Listas.
8.- Listas y cadenas.
8.2.- Evaluación de expresiones. La función eval().

- La cadena en la función **eval()** se analiza y determina como una expresión de Python. Si no puede ser evaluada como una expresión Python, generará un error.
- Este método sólo convertirá la cadena en una lista cuando la cadena sea segura.

```
>>> suma = "2 + 3"
>>> resultado = eval(suma)
>>> resultado
5

>>> colores = "['Rojo', 'Verde', 'Ambar']"
>>> cadena = eval(colores)
>>> cadena
['Rojo', 'Verde', 'Ambar']
```

UT04.01 - Estructuras de datos. Listas.
8.- Listas y cadenas.
8.3.- Tokenización de cadenas (I). El método split().

- El método split() **permite obtener una lista con sus componentes**. La lista que genera está formada por las palabras de la cadena tomando como separador el carácter indicado como parámetro.
- La sintaxis es la siguiente:

  lista = cadena.split(separador)
- Si no se indica ningún carácter se usa como separadores el blanco y el retorno de carro.
- Si el separador es espacio o retorno de carro se eliminan obteniendo la lista de todas las palabras que hay en la frase.

  Si el separador es otro carácter (por ejemplo la arroba, @), se considera una componente todo lo que se encuentra entre dos arrobas consecutivas. Por ello, si el texto contenga dos arrobas una a continuación de la otra, se devolverá una componente vacía.

```
>>> cadena = """Esta es
... una       cadena
... muy larga"""
>>> lista = cadena.split()
>>> lista
['Esta', 'es', 'una', 'cadena', 'muy', 'larga']

>>> cadena = "@@@Hola@@@@@@Mundo"
>>> lista = cadena.split("@")
>>> lista
['','','','Hola','','','','','','Mundo']
```

UT04.01 - Estructuras de datos. Listas.
8.- Listas y cadenas.
8.3.- Tokenización de cadenas (II). El método split().

- Un segundo argumento en split() indica cuál es el máximo de divisiones que puede tener lugar.
- La sintaxis es la siguiente:

  lista = cadena.split(sep, div)

- Si se indica -1 (valor por defecto) representa una cantidad ilimitada.
- El método devuelve una lista que contiene tantos elementos como divisiones se ha indicado y un elemento final con el resto de la cadena.

```
>>> cadena = "Rojo Verde Azul Marrón"
>>> lista = cadena.split(" ",2)
>>> lista
['Rojo', 'Verde', 'Azul Marrón']
```

UT04.01 - Estructuras de datos. Listas.
8.- Listas y cadenas.
8.4.- Unión de Tokens para formar una cadena. El método join().

- El método join() obtiene una cadena que se obtiene concatenando los elementos de la lista usando como separador la cadena a la que se aplica el método.
- La sintaxis es la siguiente:

  cadena = separador.join(lista)

- En el ejemplo se genera una cadena de texto usando como separador el carácter “ “ (espacio en blanco).

```
>>> lista = ["Esta", "es", "cadena", "de", "texto"]
>>> cadena = " ".join(lista)
>>> cadena
'Esta es cadena de texto'
```

UT04.01 - Estructuras de datos. Listas.
8.- Listas y cadenas.
8.5.- Partir una cadena en dos partes. El método partition().

- Un segundo método de separación es **partition()**. Este método retorna una **tupla** de tres elementos:
  - El bloque de caracteres anterior.
  - El separador.
  - El bloque posterior.
- Existe el método **rpartition()** que realiza la misma función pero desde el final de la cadena.

```
>>> cadena = "Esta es una & que nos separa"
>>> lista = cadena.partition("&")
>>> lista
('Esta es una ', '&', ' que nos separa')

>>> cadena = "Esta es una & que nos & separa"
>>> lista = cadena.rpartition("&")
>>> lista
('Esta es una & que nos ', '&', ' separa')
```