
Unidad 1

Estructuras de Almacenamiento

1º DAM

Estructuras de datos

Hasta ahora hemos estado trabajando con tipos de datos primitivos, en los que en cada variable podía guardar un valor primitivo: un entero, una cadena, etc.

Sin embargo, imaginemos que deseamos guardar las notas de una clase de 25 alumnos. ¿Creamos 25 variables?

Necesitaríamos una estructura en la que en una variable pudiéramos ir guardando la **lista** de los 25 número con las notas.

Las estructuras de datos en Python se pueden entender como un tipo de dato compuesto, debido a que en una misma variable podemos almacenar una estructura completa con información.

Las estructuras de datos más comunes en Python son:

- las listas
- las tablas
- ¡¡las cadenas!!

Listas

Una lista es una estructura de datos que nos permite almacenar varios valores de un tipo primitivo en una misma variable.

Realmente, una lista de Python es un tipo de array o tabla (un vector)

Al usar lista puede que no sepamos de primeras cuántos valores vamos a almacenar.

Las listas en Python, a diferencia de en otros lenguajes, te permiten guardar valores de diferentes tipos.

Volvamos al problema de las 25 notas de examen de una clase, nos crearemos una variable denominada `notasClase` y ahí guardaremos las 25 notas.

Listas

Declaración:

Lo primero que debemos hacer es crear esa lista y luego podremos añadir elementos.

Existen dos modos equivalentes de declarar una lista

```
lista1= []  
lista2 = list()
```

También puedo añadir elementos en la misma declaración :

```
lista1= [1, 2, True, 'Hola', 5.8]  
lista2 = list([1, 2, True, 'Hola', 5.8])
```

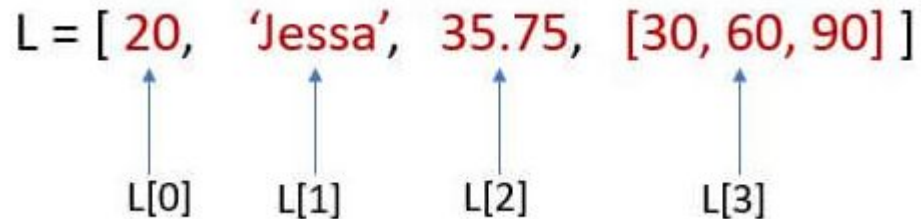
En Python dentro de una lista puedo añadir valores de distintos tipos primitivos (texto, lógicos, números, etc.)

Listas

Leer un valor:

Para esto debes saber que:

- cada elemento se guarda por orden según ha sido insertado
- el primer elemento ocupa la posición 0.
- los valores almacenados en las listas pueden ser de diferentes tipos
- puedo guardar dos veces el mismo valor.
- puedo acceder a cada valor con ***[posición]***



The diagram illustrates how to access elements in a list. It shows a list `L` containing four elements: `20`, `'Jessa'`, `35.75`, and `[30, 60, 90]`. Below each element is an index label: `L[0]`, `L[1]`, `L[2]`, and `L[3]`. Blue arrows point from each index label to its corresponding element in the list, demonstrating that the index corresponds to the position of the element in the list.

```
L = [ 20, 'Jessa', 35.75, [30, 60, 90] ]
```

`L[0]` `L[1]` `L[2]` `L[3]`

Listas

Leer un valor:

```
mi_lista = ['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana']  
print(mi_lista[0]) # Muestra Juan (la primera posición es la 0)  
print(mi_lista[1]) # Muestra Pedro  
print(mi_lista[2]) # Muestra Laura
```

El programa devolverá una excepción si intento acceder a una posición que no existe.

Recuerda que, una excepción es un error en tiempo de ejecución del programa.

Puedo indicar un valor negativo y en ese caso la lectura empezará por el final, siendo -1 el último valor insertado.

```
print(mi_lista[-1]) # Muestra Susana  
print(mi_lista[-2]) # Muestra Carmen
```

Listas

Añadir elementos de una lista:

Al agregar un elemento a una lista estamos incrementando en uno el tamaño de la lista.

1. Añadir al final de la lista → **append**

Usaremos la función `append` para agregar un elemento a una lista al final de la lista.

```
mi_lista = ['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana']  
mi_lista.append("Laura")  
mi_lista.append(False)
```

Con el siguiente comando imprimo la lista completa:

```
print(mi_lista)
```

La salida será:

```
['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana', 'Laura', False]
```

Listas

Añadir elementos de una lista:

2. Añadir un elemento en una posición concreta → insert

Usaremos la función append para agregar un elemento a una lista al final de la lista.

```
mi_lista = ['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana']  
mi_lista.insert(2, "Lucía")
```

Con el siguiente comando imprimo la lista completa:

```
print(mi_lista)
```

La salida será:

```
['Juan', 'Pedro', 'Lucía', 'Laura', 'Carmen', 'Susana']
```


Listas

Añadir elementos de una lista:

3. Concatenar listas → + o extend

También puedo añadir elementos concatenando o uniendo dos listas:

```
mi_lista = ['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana']
```

```
mi_lista = mi_lista + ["Laura", False]
```

```
print(mi_lista)
```

ó

```
mi_lista = ['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana']
```

```
mi_lista = mi_lista.extend(["Laura", False])
```

```
print(mi_lista)
```

La salida será la misma en ambos casos:

```
['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana', 'Laura', False]
```

Listas

Eliminar elementos de una lista:

Puedo eliminar elementos de dos formas:

- por posición → usando pop
- por valor → usando remove

Al eliminar un elemento, el tamaño de lista se decrementará en uno.

1. Eliminar un elemento que está en una posición

```
mi_lista = ['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana']  
print(mi_lista)  
mi_lista.pop(2)  
print(mi_lista)
```

['Juan', 'Pedro', 'Laura', 'Carmen', 'Susana']

['Juan', 'Pedro', 'Carmen', 'Susana']

Listas

Eliminar elementos de una lista:

2. Eliminar por valor → usando remove

Eliminará el **primer** elemento que encuentre con ese valor.

```
palabras = ['hola', 'hello', 'hello', 'ola']
```

```
print(palabras)
```

```
palabras.remove('hello')
```

```
print(palabras)
```

```
['hola', 'hello', 'hello', 'ola']
```

```
['hola', 'hello', 'ola']
```

Importante, si el valor no está contenido en la lista, nos devolverá una excepción:

```
palabra.remove('ello')
```

```
ValueError: list.remove(x): x not in list
```

Listas

Devolver el número de elementos de una lista:

A menudo, me conviene saber cuántos elementos tengo en mi lista.

Para ello, usaremos el método `len()`

```
palabras = ['hola', 'hello', 'hello', 'ola']
```

```
print(palabras)
```

```
print(len(palabras))
```

```
palabras.remove('hello')
```

```
print(len(palabras))
```

```
print(palabras)
```

```
['hola', 'hello', 'hello', 'ola']
```

4

3

```
['hola', 'hello', 'ola']
```

Listas

Recorrer una lista:

Utilizaremos los bucles para recorrer una lista. Para ello podremos usar diferentes sintaxis:

1. Recorrido por los elementos

```
for valorLista in mi_lista:  
    print(valorLista)
```

Este bucle imprimiría la lista completa de valores que tiene mi_lista

1. Recorrido utilizando los índices o posiciones en los que está guardado cada valor.

```
for i in range(len(edaded)) :  
    print(edaded[i])
```

Observa que para saber hasta qué valor tiene que llegar el índice usamos la función len que nos devuelve la longitud o número de elementos.

Listas

Recorrer una lista:

Es tu turno:

Adapta el bloque for con la función len para hacerlo utilizando un while en lugar de un for.

Cadenas

Una cadena puede ser considerada como una lista de caracteres.

- Puedo **acceder** a un carácter mediante su posición

```
a = "Hello, World!"
```

```
print(a[1])
```

- Generar una **subcadena**:

```
mensaje9 = "Hola Mundo"
```

```
mensaje9a = mensaje9[1:8]
```

- **Concatenar** cadenas

```
mensaje1 = 'Hola' + ' ' + 'Mundo'
```

- **Longitud** de una cadena

```
len(mensaje2a)
```

Cadenas

Una cadena puede ser considerada como una lista de caracteres.

- Recorrer los caracteres de una cadena:

```
for x in "banana":  
    print(x)
```

- Operador in

```
txt = "The best things in life are free!"  
print("free" in txt)
```

- Operador not in

```
print("expensive" not in txt)
```


Cadenas

- Buscar una subcadena

```
mensaje5 = "Hola Mundo"
```

```
mensaje5a = mensaje5.find("Mundo")
```

Si no está presente devolverá -1, si lo está, la posición en la que comienza

El método index, que vimos con las listas, es equivalente a find con la salvedad de que si no encuentra el elemento, lanza una excepción en lugar de devolver -1.

- Reemplazar una parte de la cadena:

```
mensaje8 = "HOLA MUNDO"
```

```
mensaje8a = mensaje7.replace("L", "pizza")
```

Cadenas

- Multiplicar cadenas

```
mensaje2a = 'Hola ' * 3
```

- Pasar a minúsculas

```
mensaje8.lower()
```

- Pasar a mayúsculas

```
mensaje8.upper()
```

Cadenas

Generar una lista a partir de una cadena:

```
txt = "Bienvenidos a la clase de programación"
```

```
lista = txt.split() # por defecto separa por espacios
```

```
['Bienvenidos', 'a', 'la', 'clase', 'de', 'programación']
```

```
string.split(separator, maxsplit)
```

Así puedo definir cómo separar y qué máximo de separaciones quiero hacer. Por defecto, serían todas las ocurrencias

Cadenas

Caracter \

Se pueden crear tanto con comillas simples como dobles.

Una situación que muchas veces se puede dar, es cuando queremos introducir una comilla, bien sea simple ' o doble " dentro de una cadena. Si lo hacemos de la siguiente forma tendríamos un error, ya que Python no sabe muy bien dónde empieza y termina.

```
#s = "Esto es una comilla doble " de ejemplo" # Error!
```

Para resolver este problema debemos recurrir a las secuencias de escape.

```
s = "Esto es una comilla doble \" de ejemplo"
```



