Listas en Python

🧠 ¿Qué es una lista en Python?

Una **lista** es una colección de elementos almacenados en una sola variable. Es uno de los cuatro tipos de datos integrados en Python para almacenar colecciones:

- **Lista**: ordenada, modificable, permite duplicados
- **Tupla**: ordenada, inmodificable, permite duplicados
- **Conjunto (Set)**: no ordenado, no indexado, sin duplicados
- **Diccionario**: no ordenado (aunque desde Python 3.6+ conserva el orden de inserción), pares clave-valor

---

🛠️ Crear una lista

```
Mi_lista = ["manzana", "banana", "cereza"]
print(Mi_lista)
```

Las listas se definen usando **corchetes** `[]`.

📚 Propiedades clave de las listas

Propiedad	Descripción
Ordenada	Los elementos mantienen el orden en que fueron agregados
Indexada	Cada elemento tiene una posición: `Mi_lista[0]` es `"manzana"`
Modificable	Puedes modificar, agregar o eliminar elementos
Permite duplicados	Se pueden repetir valores dentro de la lista

Ejemplo con duplicados:

```
Mi_lista = ["manzana", "banana", "cereza", "manzana", "cereza"]
print(Mi_lista)
```

✏️ Operaciones comunes con listas

Algunos métodos útiles:

```
Mi_lista.append("naranja")      # Agrega un elemento al final
Mi_lista.insert(1, "kiwi")      # Inserta en una posición específica
Mi_lista.remove("banana")       # Elimina un elemento específico
Mi_lista.pop()                  # Elimina el último elemento
Mi_lista.sort()                 # Ordena la lista alfabéticamente
Mi_lista.reverse()              # Invierte el orden de la lista
```

🧠 Diccionarios en Python

Un **diccionario** se utiliza para almacenar valores de datos en pares **clave:valor**.

Es una colección que es **ordenada** (a partir de Python 3.7), **modificable** y **no permite duplicados**.

Los diccionarios se escriben con **llaves** {}, y cada elemento tiene una clave y un valor.

🛠️ Crear y mostrar un diccionario

```
Diccionario = {

  "marca": "Ford",

  "modelo": "Mustang",

  "año": 1964
```

```
}

print(Diccionario)
```

📚 Características de los diccionarios

Propiedad	Descripción
Ordenados	Desde Python 3.7, los elementos mantienen el orden de inserción
Modificables	Se pueden cambiar, agregar o eliminar elementos
Sin duplicados	No se pueden tener dos claves iguales; si se repite una clave, se sobrescribe

---

🔑 Acceder a elementos por clave

```
Diccionario = {
  "marca": "Ford",
  "modelo": "Mustang",
  "año": 1964
}

print(Diccionario["marca"])  # Imprime "Ford"
```

🔄 ¿Ordenado o no ordenado?

- **Python 3.7 en adelante**: los diccionarios son **ordenados**.
- **Python 3.6 y versiones anteriores**: los diccionarios son **no ordenados**.

Ordenado significa que los elementos tienen un orden definido que no cambia.
No ordenado significa que no se puede acceder a los elementos por índice.

✏️ Modificable

Puedes cambiar, agregar o eliminar elementos después de crear el diccionario.

🚫 No se permiten claves duplicadas

Si se repite una clave, el valor anterior se sobrescribe:

Diccionario = {

"marca": "Ford",

"modelo": "Mustang",

"año": 1964,

"año": 2020

}

print(Diccionario) # El valor de "año" será 2020

🧠 Método `sort()` en listas de Python

El método `sort()` se utiliza para **ordenar** los elementos de una lista.

Por defecto, ordena los elementos en **orden ascendente** (alfabéticamente si son cadenas, numéricamente si son números).

🛠️ Ejemplo

Ordenar una lista alfabéticamente:

autos = ['Ford', 'BMW', 'Volvo']

autos.sort()

print(autos)

Resultado: `['BMW', 'Ford', 'Volvo']`

📚 Definición y uso

- `sort()` ordena la lista **en el lugar**, es decir, **modifica la lista original**.
- Puedes personalizar el criterio de ordenamiento usando una función.

🧪 Sintaxis

lista.sort(reverse=True|False, key=miFuncion)

📌 Valores de los parámetros

Parámetro	Descripción
`reverse`	Opcional. Si se establece en `True`, ordena la lista en orden **descendente**. Por defecto es `False`.

`key`	Opcional. Una función que especifica el **criterio de ordenamiento**.

🎯 Ejemplo con `reverse` y `key`

```
autos = ['Ford', 'BMW', 'Volvo']

autos.sort(reverse=True)  # Orden descendente

print(autos)


def longitud(palabra):

    return len(palabra)


autos.sort(key=longitud)  # Ordenar por longitud de palabra

print(autos)
```

🧠 Funciones en Python

Una **función** es un bloque de código que solo se ejecuta cuando se llama.

Puedes pasarle datos a una función, conocidos como **parámetros**.

Una función puede **devolver datos** como resultado.

🛠️ Crear una función

En Python, una función se define usando la palabra clave `def`:

```
def mi_funcion():
  print("Hola desde una función")
```

📞 Llamar a una función

Para llamar a una función, usa el nombre de la función seguido de paréntesis:

```
def mi_funcion():
  print("Hola desde una función")


mi_funcion()
```

📦 Argumentos

Puedes pasar información a las funciones como **argumentos**.

Los argumentos se especifican después del nombre de la función, dentro de los paréntesis. Puedes agregar tantos como necesites, separados por comas.

Ejemplo con un argumento (`nombre`):

```
def mi_funcion(nombre):
  print(nombre + " Rodriguez")


mi_funcion("Emil")

mi_funcion("Tobias")

mi_funcion("Linus")
```

En la documentación de Python, los argumentos suelen abreviarse como **args**.

❓ ¿Parámetros o argumentos?

Los términos **parámetro** y **argumento** pueden usarse para referirse a lo mismo: la información que se pasa a una función.

Desde la perspectiva de una función:

- Un **parámetro** es la variable que aparece entre paréntesis en la definición de la función.
- Un **argumento** es el valor que se envía a la función cuando se llama.

🔢 Número de argumentos

Por defecto, una función debe ser llamada con el número correcto de argumentos.
Si tu función espera 2 argumentos, debes llamarla con 2, ni más ni menos.

Ejemplo correcto:

```
def mi_funcion(nombre, apellido):

  print(nombre + " " + apellido)



mi_funcion("Emil", "Rodriguez")
```

Ejemplo incorrecto (solo 1 argumento):

```
def mi_funcion(nombre, apellido):

  print(nombre + " " + apellido)



mi_funcion("Emil")  # Esto generará un error
```

🧠 Condiciones y sentencias `if` en Python

Python admite las condiciones lógicas habituales de las matemáticas:

- **Igual a**: `a == b`
- **Distinto de**: `a != b`
- **Menor que**: `a < b`
- **Menor o igual que**: `a <= b`
- **Mayor que**: `a > b`
- **Mayor o igual que**: `a >= b`

Estas condiciones se pueden usar de varias formas, pero lo más común es en **sentencias `if`** y **bucles**.

🛠️ Sentencia `if`

Una sentencia `if` se escribe usando la palabra clave `if`:

```
a = 33

b = 200

if b > a:

  print("b es mayor que a")
```

En este ejemplo usamos dos variables, `a` y `b`, para comprobar si `b` es mayor que `a`. Como `b` es 200 y `a` es 33, se imprime en pantalla que “b es mayor que a”.

🔠 Identación

Python **depende de la identación** (espacios en blanco al comienzo de una línea) para definir el alcance del código. Otros lenguajes usan llaves `{}` para esto.

Ejemplo incorrecto (sin identación):

```
a = 33

b = 200

if b > a:

print("b es mayor que a")  # Esto generará un error
```

🔄 elif

La palabra clave `elif` es la forma de Python de decir: “si las condiciones anteriores no fueron verdaderas, prueba esta”.

```
a = 33

b = 33

if b > a:

  print("b es mayor que a")

elif a == b:

  print("a y b son iguales")
```

Como `a` es igual a `b`, la primera condición no se cumple, pero la condición `elif` sí, por lo que se imprime “a y b son iguales”.

🧲 else

La palabra clave `else` captura cualquier caso que no haya sido cubierto por las condiciones anteriores.

```
a = 200

b = 33

if b > a:

  print("b es mayor que a")

elif a == b:

  print("a y b son iguales")

else:

  print("a es mayor que b")
```

Aquí, como `a` es mayor que `b`, ni `if` ni `elif` se cumplen, así que se ejecuta el bloque `else`.

También puedes usar `else` sin `elif`:

```
a = 200

b = 33

if b > a:

  print("b es mayor que a")

else:

  print("b no es mayor que a")
```
