Clase 3 curso python para profesores 2026

Listas y tuplas son estructuras de datos que nos permiten almacenar colecciones de elementos. Aunque son muy similares tienen como diferencia principal que las listas son mutables (pueden cambiarse una vez declaradas) y las tuplas no.

Para crear listas y tuplas en python:

```
# Lista
mi_lista = [1, 2, 3, "hola", True]

# Tupla
mi_tupla = (1, 2, 3, "hola", True)
```

Podemos observar que en una lista o tupla no necesariamente todos los elementos son del mismo tipo. Esto hace más complicado la gestión de la tupla/lista pero nos da esa flexibilidad característica de python (¿futuros problemas?)

Cosas que podemos hacer con las listas:

Agregar elementos

```
# Agregar al final
mi_lista.append("nuevo")

# Agregar en una posición específica
mi_lista.insert(2, "insertado")

# Agregar varios elementos a la vez
mi_lista.extend([4, 5, 6])
```

Eliminar elementos

```
# Eliminar por valor (solo la primera coincidencia)
mi_lista.remove("hola")

# Eliminar por índice
del mi_lista[1]

# Eliminar y obtener el valor eliminado
valor = mi_lista.pop()      # Último
valor = mi_lista.pop(0)     # En posición 0
```

Otras operaciones útiles

```
# Saber cuántas veces aparece un valor
mi_lista.count(2)
```

```
# Saber el índice de un valor
mi_lista.index("nuevo")

# Ordenar la lista (solo si los elementos son comparables entre sí)
numeros = [3, 1, 4, 2]
numeros.sort()            # Ascendente
numeros.sort(reverse=True)  # Descendente

# Invertir el orden
numeros.reverse()

# Copiar una lista
copia = mi_lista.copy()

# Vaciar la lista
mi_lista.clear()
```

Cosas que podemos hacer con las tuplas:

Cómo crear una tupla

```
tupla = (10, 20, 30)
# También se puede crear sin paréntesis
otra = 1, 2, 3

# Tupla de un solo elemento (¡importante la coma!)
solo_uno = (5,)  # sin la coma sería solo el número 5

# Esta sintaxis nos permite realizar intercambio de valores muy fácilmente

a = 7

b = 8

a, b = b, a
```

Operaciones disponibles

Aunque no se pueden modificar, sí podemos:

```
# Acceder a elementos
print(tupla[0])

# Contar ocurrencias
print(tupla.count(10))

# Buscar índice
print(tupla.index(20))

# Longitud
len(tupla)
```

¿Por qué usar tuplas?

- Ocupan menos memoria.
- Son más rápidas que las listas.
- Se usan para datos **constantes** o que **no deben modificarse**.

Acceso a elementos y slicing (para ambas)

```
lista = [10, 20, 30, 40, 50]

# Acceso por índice
print(lista[0])      # Primer elemento
print(lista[-1])     # Último elemento

# Slicing
print(lista[1:4])    # [20, 30, 40]
print(lista[:3])     # [10, 20, 30]
print(lista[::2])    # [10, 30, 50]
```

¿Por qué necesitamos hacer copy() o lista[:] para tener dos copias distintas de una lista? (Avanzado)

El problema

Cuando haces esto:

```
lista1 = [1, 2, 3]
lista2 = lista1
```

No estás creando una nueva lista. **Estás creando una segunda variable que apunta a la misma lista en memoria.**
Es decir:

- lista1 y lista2 **son el mismo objeto**.
- Si cambias una, **la otra también cambia**.

```
lista2.append(4)
print(lista1)  # [1, 2, 3, 4]  <-- también cambió
```

¿Por qué ocurre esto?

Porque las listas son **objetos mutables** en Python, y las variables solo guardan una **referencia** a esos objetos. Al hacer lista2 = lista1, no se copia la lista, solo se copia la referencia.

¿Cómo hacer una copia independiente?

Para tener **dos listas distintas**, usamos:

1. copy()

```
lista1 = [1, 2, 3]
lista2 = lista1.copy()
lista2.append(4)

print(lista1)  # [1, 2, 3]
print(lista2)  # [1, 2, 3, 4]
```

2. lista[:] (slicing)

```
lista1 = [1, 2, 3]
lista2 = lista1[:]
lista2.append(99)

print(lista1)  # [1, 2, 3]
print(lista2)  # [1, 2, 3, 99]
```

Ejercicios:

1. Crear una lista con los números del 1 al 5.

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

2. Acceder al tercer elemento de la lista [10, 20, 30, 40, 50].

```
lista = [10, 20, 30, 40, 50]
print(lista[2])  # 30
```

3. Cambiar el valor del segundo elemento a 99.

```
lista = [10, 20, 30]
lista[1] = 99
# Resultado: [10, 99, 30]
```

4. Agregar el número 100 al final de la lista.

```
lista = [1, 2, 3]
lista.append(100)
# Resultado: [1, 2, 3, 100]
```

5. Insertar el número 50 en la segunda posición.

```
lista = [10, 20, 30]
lista.insert(1, 50)
# Resultado: [10, 50, 20, 30]
```

6. Eliminar el elemento 20 de la lista.

```
lista = [10, 20, 30]
lista.remove(20)
# Resultado: [10, 30]
```

7. Eliminar el último elemento de la lista y mostrarlo.

```
lista = [1, 2, 3]
ultimo = lista.pop()
print(ultimo)  # 3
```

8. Ordenar una lista de números en orden ascendente.

```
lista = [4, 1, 3, 2]
lista.sort()
# Resultado: [1, 2, 3, 4]
```

9. Ordenar una lista en orden descendente.

```
lista = [4, 1, 3, 2]
lista.sort(reverse=True)
# Resultado: [4, 3, 2, 1]
```

10. Contar cuántas veces aparece el número 2.

```
lista = [1, 2, 2, 3, 2]
print(lista.count(2))  # 3
```

11. Obtener el índice del número 30 en la lista [10, 20, 30, 40].

```
lista = [10, 20, 30, 40]
print(lista.index(30))  # 2
```

12. Vaciar completamente una lista.

```
lista = [1, 2, 3]
lista.clear()
# Resultado: []
```

13. Crear una copia de la lista [5, 6, 7].

```
original = [5, 6, 7]
copia = original.copy()
```

14. Invertir el orden de los elementos de una lista.

```
lista = [1, 2, 3]
lista.reverse()
# Resultado: [3, 2, 1]
```

15. Crear una tupla con los valores "a", "b", "c".

```
tupla = ("a", "b", "c")
```

16. Acceder al segundo elemento de una tupla.

```
tupla = (10, 20, 30)
print(tupla[1])  # 20
```

17. Contar cuántas veces aparece el número 1 en una tupla.

```
tupla = (1, 2, 1, 3, 1)
print(tupla.count(1))  # 3
```

18. Obtener el índice del valor "python" en una tupla.

```
tupla = ("java", "python", "c++")
print(tupla.index("python"))  # 1
```

19. Usar slicing para obtener los tres primeros elementos de la lista [100, 200, 300, 400, 500].

```
lista = [100, 200, 300, 400, 500]
print(lista[:3])  # [100, 200, 300]
```

20. Crear una tupla con un solo elemento (el número 5).

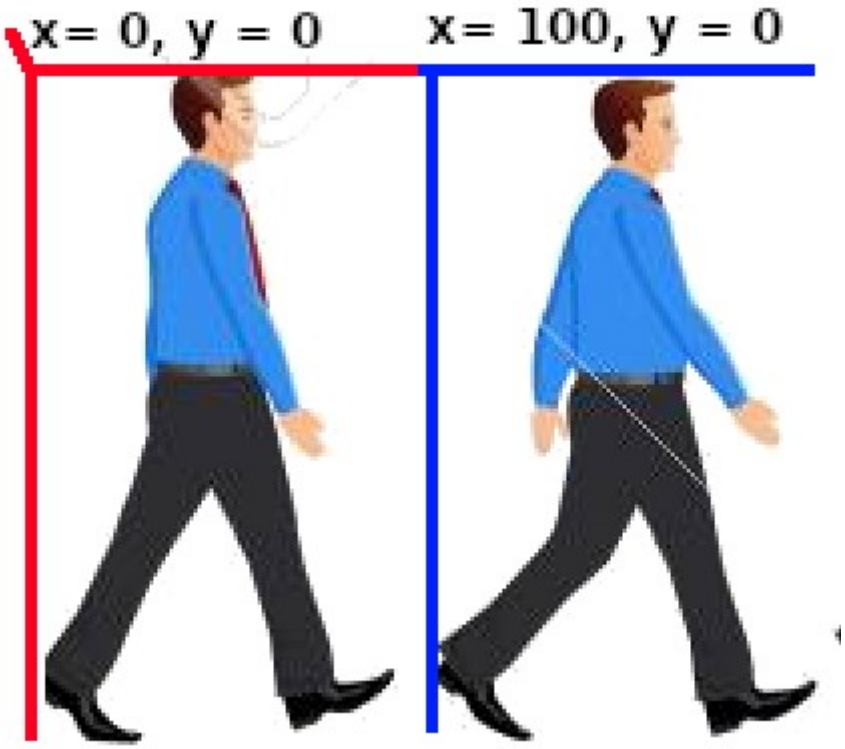
```
tupla = (5,)  # ¡No olvidar la coma!
```

Antes de continuar vamos a explicar las coordenadas que evamos a utilizar en los siguientes scripts

Podemos observar que la lista (o tupla) movbicho del hombreAndando.png tiene estos valores:

```
movbicho = ((0,0,90,175),(100,0,90,175),(200,0,90,175),(300,0,90,175),
(400,0,90,175),(500,0,90,175),(600,0,90,175),(700,0,90,175),
(800,0,90,175))
```

Vamos a fijarnos en las dos primeras coordenadas y observemos la selección con notas que hemos hecho de la imagen:



Cuando la variable fase_mov tiene el valor 0, movbicho[0] será igual a `(0,0,90,175)`

Cuando la variable fase_mov pasa al valor 1, movbicho[1] será igual a `(100,0,90,175)`

Entonces, al realizar el “dibujo” del personaje

Los valores 90, 175 son el ancho y el largo del rectángulo que surgiría desde las coordenadas (0,0) y (100,0), donde la primera coordenada nos indica la posición x (la hemos llamado margen en algunos script, y la segunda posición sería la posición y.

Entonces al realizar el “dibujo” del personaje en la línea:

```
visor.blit(bicho, (pos,100), movbicho[fase_mov])
```

Obtendremos el rectángulo que muestra esa fase del movimiento. ¿Puedes realizar esto mismo con gimp (añadir líneas guía) y la imagen de Ken? ¿Puedes poner líneas guías verticales en las coordenadas indicadas en la primera posición de cada una de las tuplas de movbicho?

Si cada movimiento del personaje se encuentra en posiciones que nos permitan realizar módulo el código es mucho más fácil, aunque algo más difícil de entender. Puedes ver un ejemplo en los códigos: personaje_movimiento_con_modulo.py

Otra forma sencilla de mostrar cómo el personaje pasa por distintos escenarios es mediante la creación de una lista/tupla en la que cada posición es la imagen o color de un escenario diferente. Basta con añadir una variable que controle la posición del escenario y que aumente si el personaje se pasa por la derecha, y disminuyéndolo si el personaje pasa por la izquierda. Con esta variable, utilizando **visor.fill()** ya estaría. Pues ese es el **RETO**: crea una lista/tupla con escenarios y que vayan cambiando según el personaje salga de la pantalla