Modulo 4: Tuplas

Tipos de secuencias y mutabilidad

Antes de comenzar a hablar acerca de **tuplas y diccionarios**, se deben introducir dos conceptos importantes: **tipos de secuencia y mutabilidad**.

Un **tipo de secuencia es un tipo de dato en Python el cual es capaz de almacenar más de un valor (o ninguno si la secuencia esta vacía), los cuales pueden ser secuencialmente (de ahí el nombre) examinados**, elemento por elemento.

Debido a que el bucle **for** es una herramienta especialmente diseñada para iterar a través de las secuencias, podemos definirlas de la siguiente manera: **una secuencia es un tipo de dato que puede ser escaneado por el bucle for**.

Hasta ahora, has trabajado con una secuencia en Python, la lista. La lista es un clásico ejemplo de una secuencia de Python. Aunque existen otras secuencias dignas de mencionar, las cuales se presentaran a continuación.

La segunda noción - **la mutabilidad** - es una propiedad de cualquier tipo de dato en Python que describe su disponibilidad para poder cambiar libremente durante la ejecución de un programa. Existen dos tipos de datos en Python: **mutables e inmutables**.

Los datos mutables pueden ser actualizados libremente en cualquier momento, a esta operación se le denomina «in situ».

In situ es una expresión en Latín que se traduce literalmente como *en posición*, en el lugar o momento. Por ejemplo, la siguiente instrucción modifica los datos «in situ»:

```
list.append(1)
```

Los datos inmutables no pueden ser modificados de esta manera.

Imagina que una lista solo puede ser asignada y leída. No podrías adjuntar ni remover un elemento de la lista. Si se agrega un elemento al final de la lista provocaría que la lista se cree desde cero.

Se tendría que crear una lista completamente nueva, la cual contenga los elementos ya existentes más el nuevo elemento.

El tipo de datos que se desea tratar ahora se llama **tupla**. **Una tupla es una secuencia inmutable**. Se puede comportar como una lista pero no puede ser modificada en el momento.

¿Qué es una tupla?

Lo primero que distingue una lista de una tupla es la sintaxis empleada para crearlas. Las **tuplas utilizan paréntesis**, mientras que las listas usan corchetes, aunque también es **posible crear una tupla tan solo separando los valores por comas**.

Observa el ejemplo:

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125
```

Se definieron dos tuplas, ambas contienen cuatro elementos.

A continuación se imprimen en consola:

```
tuple_1 = (1, 2, 4, 8)
tuple_2 = 1., .5, .25, .125

print(tuple_1)
print(tuple_2)
```

Esto es lo que se muestra en consola:

```
(1, 2, 4, 8)
(1.0, 0.5, 0.25, 0.125)
```

Nota: **cada elemento de una tupla puede ser de distinto tipo** (punto flotante, entero, cadena, o cualquier otro tipo de dato).

¿Cómo crear una tupla?

¿Es posible crear una tupla vacía? Si, solo se necesitan unos paréntesis:

```
empty_tuple = ()
```

Si se desea crear una tupla de un solo elemento, se debe de considerar el hecho de que, debido a la sintaxis (una tupla debe de poder distinguirse de un valor entero ordinario), se debe de colocar una coma al final:

```
one_element_tuple_1 = (1, )
one_element_tuple_2 = 1.,
```

El quitar las comas no arruinará el programa en el sentido sintáctico, pero serán dos variables, no tuplas.

¿Cómo utilizar un tupla?

Si deseas leer los elementos de una tupla, lo puedes hacer de la misma manera que se hace con las listas.

```
my_tuple = (1, 10, 100, 1000)

print(my_tuple[0])
print(my_tuple[-1])
print(my_tuple[1:])
print(my_tuple[:-2])

for elem in my_tuple:
    print(elem)
```

El programa debe de generar la siguiente salida, ejecútalo y comprueba:

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

```
1
10
100
1000
```

Las similitudes pueden ser engañosas - **no intentes modificar el contenido de la tupla** ¡No es una lista!

Todas estas instrucciones (con excepción de primera) causarán un error de ejecución.

¿Qué más pueden hacer las tuplas?

- La función len() acepta tuplas, y regresa el número de elementos contenidos dentro.
- El operador + puede unir tuplas (ya se ha mostrado esto antes).
- El operador * puede multiplicar las tuplas, así como las listas.
- Los operadores in y not in funcionan de la misma manera que en las listas.

```
my_tuple = (1, 10, 100)

t1 = my_tuple + (1000, 10000)
t2 = my_tuple * 3

print(len(t2))
print(t1)
print(t2)
print(10 in my_tuple)
print(-10 not in my_tuple)
```

La salida es la siguiente:

```
9
(1, 10, 100, 1000, 10000)
(1, 10, 100, 1, 10, 100, 1, 10, 100)
True
True
```

Una de las propiedades de las tuplas más útiles es que pueden **aparecer en el lado izquierdo del operador de asignación**. Este fenómeno ya se vio con anterioridad, cuando fue necesario encontrar una manera de intercambiar los valores entre dos variables.

Observa el siguiente fragmento de código:

```
var = 123

t1 = (1, )
t2 = (2, )
t3 = (3, var)

t1, t2, t3 = t2, t3, t1

print(t1, t2, t3)
```

Muestra tres tuplas interactuando en efecto, los valores almacenados en ellas «circulan» entre ellas. t1 se convierte en t2, t2 se convierte en t3, y t3 se convierte en t1.

Nota: el ejemplo presenta un importante hecho mas: los **elementos de una tupla pueden ser variables**, no solo literales. Además, pueden ser expresiones si se encuentran en el lado derecho del operador de asignación.

Puntos Clave: Tuplas

1. Las Tuplas son colecciones de datos ordenadas e inmutables. Se puede pensar en ellas como listas inmutables. Se definen con paréntesis:

```
my_tuple = (1, 2, True, "una cadena", (3, 4), [5, 6], None)
print(my_tuple)

my_list = [1, 2, True, "una cadena", (3, 4), [5, 6], None]
print(my_list)
```

Cada elemento de la tupla puede ser de un tipo de dato diferente (por ejemplo, enteros, cadenas, boleanos, etc.). Las tuplas pueden contener otras tuplas o listas (y viceversa).

2. Se puede crear una tupla vacía de la siguiente manera:

```
empty_tuple = ()
print(type(empty_tuple))    # salida: <class 'tuple'>
```

3. La tupla de un solo elemento se define de la siguiente manera:

```
one_elem_tuple_1 = ("uno", )    # Paréntesis y una coma.
one_elem_tuple_2 = "uno",       # Sin paréntesis, solo la coma.
```

Si se elimina la coma, Python creará una variable no una tupla:

```
my_tuple_1 = 1,
print(type(my_tuple_1))    # salida: <class 'tuple'>

my_tuple_2 = 1             # Esto no es una tupla.
print(type(my_tuple_2))    # salida: <class 'int'>
```

4. Se pueden acceder los elementos de la tupla al indexarlos:

```
my_tuple = (1, 2.0, "cadena", [3, 4], (5, ), True)
print(my_tuple[3])    # salida: [3, 4]
```

5. Las tuplas son immutable, lo que significa que no se puede agregar, modificar, cambiar o quitar elementos. El siguiente fragmento de código provocará una excepción:

```
my_tuple = (1, 2.0, "cadena", [3, 4], (5, ), True)
my_tuple[2] = "guitarra"    # La excepción TypeError será lanzada.
```

Sin embargo, se puede eliminar la tupla completa:

```
my_tuple = 1, 2, 3,
del my_tuple
print(my_tuple)    # NameError: name 'my_tuple' is not defined
```

6. Puedes iterar a través de los elementos de una tupla con un bucle (Ejemplo 1), verificar si un elemento o no esta presente en la tupla (Ejemplo 2), emplear la función len() para verificar cuantos elementos existen en la tupla (Ejemplo 3), o incluso unir o multiplicar tuplas (Ejemplo 4):

```
# Ejemplo 1
```

```
tuple_1 = (1, 2, 3)
for elem in tuple_1:
    print(elem)

# Ejemplo 2
tuple_2 = (1, 2, 3, 4)
print(5 in tuple_2)
print(5 not in tuple_2)

# Ejemplo 3
tuple_3 = (1, 2, 3, 5)
print(len(tuple_3))

# Ejemplo 4
tuple_4 = tuple_1 + tuple_2
tuple_5 = tuple_3 * 2

print(tuple_4)
print(tuple_5)
```

EXTRA

También se puede crear una tupla utilizando la función integrada de Python **tuple()**. Esto es particularmente útil cuando se desea convertir un iterable (por ejemplo, una lista, rango, cadena, etcétera) en una tupla:

```
my_tuple = tuple((1, 2, "cadena"))
print(my_tuple)

my_list = [2, 4, 6]
print(my_list)    # salida: [2, 4, 6]
print(type(my_list))    # salida: <class 'list'>
tup = tuple(my_list)
print(tup)    # salida: (2, 4, 6)
print(type(tup))    # salida: <class 'tuple'>
```

De la misma manera, cuando se desea convertir un iterable en una lista, se puede emplear la función integrada de Python denominada **list()**:

```
tup = 1, 2, 3,
my_list = list(tup)
print(type(my_list))    # salida: <class 'list'>
```