# Tuplas

Son unas colecciones muy parecidas a las listas con la peculiaridad de que son inmutables:

tupla = (100,"Hola",[1,2,3],-50)

tupla

(100, 'Hola', [1, 2, 3], -50)

## Indexación y slicing

print(tupla)

print(tupla[0])

print(tupla[-1])

print(tupla[2:])

print(tupla[2][-1])

(100, 'Hola', [1, 2, 3], -50)

100

-50

([1, 2, 3, 4], -50)

4

## Inmutabilidad

tupla[0] = 50

---------------------------------------------------------------------------

TypeError Traceback (most recent call last)

<ipython-input-9-b45433b4cee9> in <module>()

----> 1 tupla[0] = 50

TypeError: 'tuple' object does not support item assignment

## Función len()

Igual que si fuera una lista podemos utilizarla para saber la longitud de una tupla:

len(tupla)

4

len(tupla[2])

3

## Métodos integrados

### **index()**

Sirve para buscar un elemento y saber su posición en la tupla:

tupla.index(100)

0

tupla.index('Hola')

1

Da error si no se encuentra:

tupla.index('Otro')

---------------------------------------------------------------------------

ValueError Traceback (most recent call last)

<ipython-input-18-640d616163a2> in <module>()

----> 1 tupla.index('Otro')

ValueError: tuple.index(x): x not in tuple

### **count()**

Sirve para contar cuantas veces aparece un elemento en una tupla:

tupla.count(100)

1

tupla.count('Algo')

0

tupla = (100,100,100,50,10)

tupla.count(100)

3

### **append() ?**

Al ser inmutables, las tuplas **no disponen** de métodos para modificar su contenido:

tupla.append(10)

---------------------------------------------------------------------------

AttributeError Traceback (most recent call last)

<ipython-input-23-758d195ec9d7> in <module>()

----> 1 tupla.append(10)

AttributeError: 'tuple' object has no attribute 'append'

# Conjuntos

Son colecciones desordenadas de elementos únicos utilizados para hacer pruebas de pertenencia a grupos y eliminación de elementos duplicados.

Para definir un conjunto vacío hay que llamar a su clase set (conjunto en inglés):

conjunto = set()

conjunto

set()

Sin embargo si lo creamos con algunos datos se definen entre llaves:

conjunto = {1,2,3}

conjunto

{1, 2, 3}

## Método add()

Sirve para añadir elementos al conjunto, pero si un elemento ya se encuentra no se añadirá de nuevo:

conjunto.add(4)

conjunto

{1, 2, 3, 4}

conjunto.add(0)

conjunto

{0, 1, 2, 3, 4}

## Colecciones desordenadas

Se dice que son desordenados porque gestionan automáticamente la posición de sus elementos, en lugar de conservarlos en la posición que nosotros los añadimos:

conjunto.add('H')

conjunto.add('A')

conjunto.add('Z')

conjunto

{0, 1, 2, 3, 4, 'A', 'Z', 'H'}

## Pertenencia a grupos

grupo = {'Hector','Juan','Mario'}

Es fácil saber si un elemento se encuentra en un conjunto utilizando la sintaxis **in**. Se utiliza mucho para trabajar con grupos:

'Hector' in grupo

True

También se puede hacer la comprobación inversa con **not in**:

'Hector' not in grupo

False

## Elementos únicos

Los conjuntos no pueden tener el mismo elemento más de una vez, se borran los duplicados automáticamente:

test = {'Hector','Hector','Hector'}

test

{'Hector'}

## Conversiones con listas

Es muy útil transformar listas a conjuntos para borrar los elementos duplicados automáticamente y viceversa:

lista = [1,2,3,3,2,1]

print(lista)

conjunto = set(lista)

lista = list(conjunto)

print(lista)

[1, 2, 3, 3, 2, 1]

[1, 2, 3]

La conversión se puede hacer en una línea:

lista = [1,2,3,3,2,1]

print(lista)

lista = list(set(lista))

print(lista)

[1, 2, 3, 3, 2, 1]

[1, 2, 3]

## Conversiones con cadenas

Hacer esta transformación sirve para crear un conjunto con todos los caracteres de la cadena, pero sin duplicados:

cadena = "Al pan pan y al vino vino"

set(cadena)

{' ', 'A', 'a', 'i', 'l', 'n', 'o', 'p', 'v', 'y'}

# Métodos de los conjuntos

## Métodos básicos

### **add()**

Añade un ítem a un conjunto, si ya existe no lo añade:

c = set()

c.add(1)

c.add(2)

c.add(3)

c

{1, 2, 3}

### **discard()**

Borra un ítem de un conjunto:

c.discard(1)

c

{2, 3}

### **copy()**

Devuelva una copia de un conjunto, ya que éstos como la mayoría de colecciones se almacenan por referencia:

c1 = {1,2,3,4}

c2 = c.copy()

print(c1, c2)

c2.discard(4)

print(c1, c2)

{1, 2, 3, 4} {1, 2, 3, 4}

{1, 2, 3, 4} {1, 2, 3}

### **clear()**

Borra todos los ítems de un conjunto:

c2.clear()

c2

set()

## Comparación de conjuntos

c1 = {1,2,3}

c2 = {3,4,5}

c3 = {-1,99}

c4 = {1,2,3,4,5}

### **isdisjoint()**

Comprueba si el conjunto es disjunto de otro conjunto, es decir, si no hay ningún elemento en común entre ellos:

c1.isdisjoint(c2)

False

### **issubset()**

Comprueba si el conjunto es subconjunto de otro conjunto, es decir, si sus ítems se encuentran todos dentro de otro:

c3.issubset(c4)

False

### **issuperset()**

Comprueba si el conjunto es contenedor de otro subconjunto, es decir, si contiene todos los ítems de otro:

c3.issuperset(c1)

False

## Métodos avanzados

Se utilizan para realizar uniones, diferencias y otras operaciones avanzadas entre conjuntos.

Suelen tener dos formas, la normal que **devuelve** el resultado, y otra que hace lo mismo pero **actualiza** el propio resultado.

### **union()**

Une un conjunto a otro y devuelve el resultado en un nuevo conjunto:

c1 = {1,2,3}

c2 = {3,4,5}

c3 = c1.union(c2)

print(c1, "+", c2, "=", c3)

{1, 2, 3} + {3, 4, 5} = {1, 2, 3, 4, 5}

### **update()**

Une un conjunto a otro en el propio conjunto:

c1.update(c2)

c1

{1, 2, 3, 4, 5}

### **difference()**

Encuentra los elementos no comunes entre dos conjuntos:

c1 = {1,2,3}

c2 = {3,4,5}

c1.difference(c2)

{1, 2}

### **difference\_update()**

Guarda en el conjunto los elementos no comunes entre dos conjuntos:

c1.difference\_update(c2)

c1

{1, 2}

### **intersection()**

Devuelve un conjunto con los elementos comunes en dos conjuntos:

c1 = {1,2,3}

c2 = {3,4,5}

c1.intersection(c2)

{3}

### **intersection\_update()**

Guarda en el conjunto los elementos comunes entre dos conjuntos:

c1.intersection\_update(c2)

c1

{3}

### **symmetric\_difference()**

Devuelve los elementos simétricamente diferentes entre dos conjuntos, es decir, todos los elementos que no concuerdan entre los dos conjuntos:

c1 = {1,2,3}

c2 = {3,4,5}

c1.symmetric\_difference(c2)

{1, 2, 4, 5}

# Diccionarios

Son junto a las listas las colecciones más utilizadas y se basan en una estructura mapeada donde cada elemento de la colección se encuentra identificado con una clave única, por lo que no puede haber dos claves iguales. En otros lenguajes se conocen como arreglos asociativos.

Los diccionarios se definen igual que los conjuntos, utilizando llaves, pero también se pueden crear vacíos con ellas:

vacio = {}

vacio

{}

Si consultamos el tipo de la variable que contiene un diccionario con la función type() encontraremos la palabra dict, esa es la clase que define los diccionarios:

type(vacio)

dict

## Definición

Para cada elemento se define la estructura **clave:valor**:

colores = {'amarillo':'yellow','azul':'blue'}

colores

{'amarillo': 'yellow', 'azul': 'blue'}

Para consultar el valor de una clave utilizaremos la clave a modo de índice:

colores['amarillo']

'yellow'

## Mutabilidad

Los diccionarios son mutables, por lo que se les puede añadir elementos sobre la marcha a través de las claves:

colores['verde'] = 'green'

colores

{'amarillo': 'yellow', 'azul': 'blue', 'verde': 'green'}

Como los diccionarios son mutables también podemos sobreescribir un valor:

colores['amarillo'] = 'white'

colores

{'amarillo': 'white', 'azul': 'blue', 'verde': 'green'}

## Función del()

Sirve para borrar un elemento del diccionario:

del(colores['amarillo'])

colores

{'azul': 'blue', 'verde': 'green'}

Por cierto, las claves también pueden ser números, pero son un poco confusas:

numeros = {10:'diez',20:'veinte'}

numeros[10]

'diez'

Una utilidad de los diccionarios es que podemos trabajar directamente con sus registros como si fueran variables:

edades = {'Hector':27,'Juan':45,'Maria':34}

edades['Hector']+=1

edades

{'Hector': 28, 'Juan': 45, 'Maria': 34}

edades['Juan'] + edades['Maria']

79

## Lectura secuencial

Es posible utilizar iteraciones **for** para recorrer los elementos del diccionario:

edades = {'Hector':27,'Juan':45,'Maria':34}

for edad in edades:

print(edad)

Maria

Hector

Juan

El problema es que se devuelven las claves en lugar de los valores. Para solucionarlo deberíamos indicar la clave del diccionario para cada elemento:

for clave in edades:

print(edades[clave])

34

27

45

Si queremos mostrar tanto la clave como el valor podríamos hacerlo así:

for clave in edades:

print(clave,edades[clave])

Maria 34

Hector 27

Juan 45

El **método .items()** nos facilita la lectura en clave y valor de los elementos. Devuelve ambos valores en cada iteración automáticamente y nos permite almacenarlos:

for clave, valor in edades.items():

print(clave, valor)

Maria 34

Hector 27

Juan 45

## Listas de diccionarios

Podemos crear nuestras propias estructuras avanzadas mezclando ambas colecciones. Mientras los diccionarios se encargarían de manejar las propiedades individuales de los registros, las listas nos permitirían manejarlos todos en conjunto:

personajes = []

gandalf = {'Nombre':'Gandalf','Clase':'Mago','Raza':'Humano'}

legolas = {'Nombre':'Legolas','Clase':'Arquero','Raza':'Elfo'}

gimli = {'Nombre':'Gimli','Clase':'Guerrero','Raza':'Enano'}

personajes.append(gandalf)

personajes.append(legolas)

personajes.append(gimli)

print(personajes)

[{'Clase': 'Mago', 'Nombre': 'Gandalf', 'Raza': 'Humano'},

{'Clase': 'Arquero', 'Nombre': 'Legolas', 'Raza': 'Elfo'},

{'Clase': 'Guerrero', 'Nombre': 'Gimli', 'Raza': 'Enano'}]

Como ahora tenemos una estructura común a través de diccionarios, podemos suponer que cada diccionario es un personaje y mostrar los registros mientras los recorremos dinámicamente con un for:

for pesonaje in personajes:

print(pesonaje['Nombre'], pesonaje['Clase'], pesonaje['Raza'])

Gandalf Mago Humano

Legolas Arquero Elfo

Gimli Guerrero Enano

# Métodos de los diccionarios

colores = { "amarillo":"yellow", "azul":"blue", "verde":"green" }

## get()

Busca un elemento a partir de su clave y si no lo encuentra devuelve un valor por defecto:

colores.get('negro','no se encuentra')

'no se encuentra'

## keys()

Genera una lista en clave de los registros del diccionario:

colores.keys()

dict\_keys(['amarillo', 'azul', 'verde'])

## values()

Genera una lista en valor de los registros del diccionario:

colores.values()

dict\_values(['yellow', 'blue', 'green'])

## items()

Genera una lista en clave-valor de los registros del diccionario:

colores.items()

dict\_items([('amarillo', 'yellow'), ('azul', 'blue'), ('verde', 'green')])

for clave, valor in colores.items():

print(clave, valor)

amarillo yellow

azul blue

verde green

## pop()

Extrae un registro de un diccionario a partir de su clave y lo borra, acepta valor por defecto:

colores.pop("amarillo", "no se ha encontrado")

'yellow'

colores.pop("negro","no se ha encontrado")

'no se ha encontrado'

## clear()

Borra todos los registros de un diccionario:

colores.clear()

colores

{}