# **Colecciones de datos**

Python integra un montón de colecciones para manejar datos. Ya conocemos las listas pero hay otros tipos que sirven para cubrir otras necesidades...

# **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'}

# **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

**Curiosidades**

Un alumno me dijo que Gandalf no es humano, disculpadme por tal ofensa 🙇‍🙇‍🙇‍

# **Pilas**

Son colecciones de elementos ordenados que únicamente permiten dos acciones:

* Añadir un elemento a la pila.
* Sacar un elemento de la pila.

La peculiaridad es que el último elemento en entrar es el primero en salir. En inglés se conocen como estructuras LIFO (Last In First Out).

Las podemos crear como listas normales y añadir elementos al final con el append():

pila = [3,4,5]

pila.append(6)

pila.append(7)

print(pila)

[3, 4, 5, 6, 7]

Para sacar los elementos utilizaremos el método pop(). Al utilizareste método devolveremos el último elemento, pero también lo borraremos:

print(pila.pop())

print(pila)

7

[3, 4, 5, 6]

Si queremos trabajar con él deberíamos asignarlo a una variable:

numero = pila.pop()

print(numero)

6

Si vamos sacando elementos llegará un momento en que la pila estará vacía y dará error porque no podrá sacar nada más:

pila.pop()

pila.pop()

pila.pop()

pila.pop()

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

IndexError Traceback (most recent call last)

<ipython-input-14-3900970cfbef> in <module>()

----> 1 pila.pop()

IndexError: pop from empty list

# **Colas**

Son colecciones de elementos ordenados que únicamente permiten dos acciones:

* Añadir un elemento a la cola.
* Sacar un elemento de la cola.

La peculiaridad es que el primer elemento en entrar es el primero en salir. En inglés se conocen como estructuras FIFO (First In First Out).

Debemos importar la colección deque manualmente para crear una cola:

from collections import deque

cola = deque()

print(cola)

deque([])

Podemos añadir elementos al crear la cola pasándolos en una lista:

cola = deque(['Hector','Juan','Miguel'])

print(cola)

deque(['Hector', 'Juan', 'Miguel'])

Luego podemos seguir añadiéndolos utilizando el método append():

cola.append('Maria')

cola.append('Arnaldo')

print(cola)

deque(['Hector', 'Juan', 'Miguel', 'Maria', 'Arnaldo'])

La parte interesante es a la hora de sacar los elementos, pues en esta ocasión utilizaremos el método popleft(). Hace lo mismo que pop() pero los extrae por la parte izquierda, que sería el principio de la cola:

print(cola.popleft())

print(cola)

'Hector'

deque(['Juan', 'Miguel', 'Maria', 'Arnaldo'])

Además al igual que antes debemos asegurarnos de almacenar los elementos al sacarlos o los perderemos:

persona = cola.popleft()

print(persona)

print(cola)

'Juan'

deque(['Miguel', 'Maria', 'Arnaldo'])