5. Estructuras de datos

Las estructuras de datos son una forma de organizar y almacenar datos en un programa de Python. En esta sección, se explicarán las cuatro estructuras de datos principales en Python: listas, tuplas, diccionarios y conjuntos.

a) Listas

Las listas son una de las estructuras de datos más utilizadas en Python. Una lista es una colección de elementos ordenados que pueden ser de cualquier tipo de datos, como enteros, flotantes, cadenas, etc. En esta sección, exploraremos cómo crear listas, realizar operaciones en ellas y usar métodos específicos para manipularlas.

b) Creación e inicialización de listas

Para crear una lista, se utilizan corchetes [ ] y se separan los elementos con comas. Por ejemplo, la siguiente línea de código crea una lista de nombres:

```
nombres = ["Ana", "Juan", "María", "Luis"]
```

También podemos crear una lista vacía y agregar elementos más tarde:

```
numeros = []
numeros.append(1)
numeros.append(2)
numeros.append(3)
```

c) Operaciones de listas

Las listas pueden ser manipuladas con una serie de operaciones. Las más comunes incluyen:

Concatenación

Podemos unir dos listas utilizando el operador +:

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
```

```
lista3 = lista1 + lista2

print(lista3)
```

```
> [1, 2, 3, 4, 5, 6]
```

Repetición

Podemos repetir una lista utilizando el operador *:

```
lista = [1, 2, 3]
lista_repetida = lista * 3

print(lista_repetida)
```

```
> [1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Segmentación

Podemos acceder a un segmento de una lista utilizando el operador **[inicio:fin]**:

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

print(segmento)
```

```
> [2, 3, 4]
```

Indexación

Podemos acceder a un elemento específico de una lista utilizando su índice:

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

print(segundo_elemento)
```

```
> 2
```

d) Métodos de listas

Además de las operaciones básicas de las listas, Python también proporciona una serie de métodos específicos para manipular listas. Algunos de los métodos más comunes incluyen:

append

Agrega un elemento al final de la lista:

```
lista = [1, 2, 3]
lista.append(4)

print(lista)
```

```
> [1, 2, 3, 4]
```

insert

Agrega un elemento en una posición específica de la lista:

```
lista = [1, 2, 3]
lista.insert(1, "hola")

print(lista)
```

```
> [1, "hola", 2, 3]
```

remove

Elimina el primer elemento con un valor específico de la lista:

```
lista = [1, 2, 3, 2]
lista.remove(2)

print(lista)
```

```
> [1, 2, 2]
```

pop

Elimina y devuelve el elemento en una posición específica de la lista:

```
lista = [1, 2, 3]
elemento_eliminado = lista.pop(1)

print(lista)
> [1, 3]

print(elemento_eliminado)
```

```
> 2
```

Además de estos métodos, existen muchos otros que se pueden utilizar con listas en Python. Es importante conocerlos para poder trabajar de manera efectiva con listas y poder sacar el máximo provecho de ellas.

e) Tuplas

Las tuplas son una estructura de datos muy similar a las listas, con la diferencia principal de que son inmutables, es decir, una vez que se crea una tupla, no se pueden modificar sus elementos. Esto las hace ideales para almacenar elementos que no deben cambiar.

Para crear una tupla, se utilizan paréntesis en lugar de corchetes como en el caso de las listas. La sintaxis es la siguiente:

```
tupla = (elemento1, elemento2, elemento3, ...)
```

f) Creación e inicialización de tuplas

Se pueden crear tuplas de varias maneras, incluyendo la declaración de una variable como una tupla vacía, o la asignación de varios valores separados por comas a una variable de tupla.

Ejemplo

```
tupla_vacia = ()
tupla_uno = (1,)
# la coma es necesaria para distinguir una tupla de un entero
tupla_varios = (1, 2, 3)
```

g) Operaciones de tuplas

Las tuplas también admiten las operaciones de concatenación, repetición, segmentación e indexación. Estas operaciones funcionan de manera similar a las listas. A continuación se describen brevemente:

Concatenación

Se realiza utilizando el operador **+**. Por ejemplo, si tenemos dos tuplas *tupla1* y *tupla2*, podemos concatenarlas de la siguiente manera:

```
tupla1 = (1, 2, 3)
tupla2 = (4, 5, 6)
tupla_concatenada = tupla1 + tupla2
print(tupla_concatenada)
```

```
> (1, 2, 3, 4, 5, 6)
```

Repetición

Se realiza utilizando el operador *. Por ejemplo, si tenemos una tupla tupla1 y queremos repetirla tres veces, podemos hacer lo siguiente:

```
tupla_repetida = tupla1 * 2
print(tupla_repetida)
```

```
> (1, 2, 3, 1, 2, 3)
```

Segmentación

Se realiza utilizando la misma sintaxis que en el caso de las listas. Por ejemplo, si tenemos una tupla *tupla1* y queremos obtener una sub-tupla que contenga los elementos del índice 1 al 3, podemos hacer lo siguiente:

```
tupla_segmentada = tupla1[1:3]
print(tupla_segmentada)
```

```
>(2, 3)
```

Indexación

Se realiza de la misma forma que en el caso de las listas. Por ejemplo, si tenemos una tupla *tupla1* y queremos obtener el elemento del índice 0, podemos hacer lo siguiente:

```
elemento1 = tupla1[0]
print(elemento1)
```

```
> 1
```

En resumen, las tuplas son una estructura de datos muy útil para almacenar elementos que no deben cambiar. Aunque son similares a las listas, se diferencian en que son **inmutables**. Además, admiten las mismas operaciones de concatenación, repetición, segmentación e indexación que las listas

h) Conjuntos

Los conjuntos, o sets en inglés, son una estructura de datos en Python que se utilizan para almacenar elementos únicos sin orden definido. Es decir, no hay elementos repetidos y el orden de los elementos no importa.

i) Creación e inicialización de conjuntos

Para crear un conjunto en Python, se utiliza la función set() y se le pasan los elementos dentro de corchetes.

```
conjunto = set([1, 2, 3, 4, 5])
print(conjunto)
```

```
> {1, 2, 3, 4, 5}
```

También se puede crear un conjunto utilizando llaves **{}** y separando los elementos por comas.

```
conjunto = {1, 2, 3, 4, 5}
print(conjunto)
```

```
> {1, 2, 3, 4, 5}
```

j) Operaciones de conjuntos

Unión

Se utiliza el operador **|** o la función **union()** para unir dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
union = conjunto1 | conjunto2
```

```
print(union)
```

```
> {1, 2, 3, 4, 5}
```

Intersección

Se utiliza el operador **&** o la función **intersection()** para obtener los elementos comunes en dos conjuntos.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
interseccion = conjunto1 & conjunto2
print(interseccion)
```

```
> {3}
```

Diferencia

Se utiliza el operador **-** o la función **difference()** para obtener los elementos que están en el primer conjunto pero no en el segundo.

```
conjunto1 = {1, 2, 3}
conjunto2 = {3, 4, 5}
diferencia = conjunto1 - conjunto2
print(diferencia)
```

```
> {1, 2}
```

k) Métodos de conjuntos

add()

Agrega un elemento al conjunto.

```
conjunto = {1, 2, 3}
conjunto.add(4)
print(conjunto)
```

```
> {1, 2, 3, 4}
```

remove()

Elimina un elemento del conjunto.

```
conjunto = {1, 2, 3, 5, 9}
conjunto.remove(5)
print(conjunto)
```

```
> {1, 2, 3, 9}
```

discard()

Elimina un elemento del conjunto, si existe.

```
conjunto = {1, 2, 3}
conjunto.discard(3)
print(conjunto)
```

```
> {1, 2}
```

En resumen, los conjuntos son una estructura de datos en Python que permiten almacenar elementos únicos sin orden definido. Se pueden realizar operaciones de unión, intersección y diferencia, y se pueden utilizar métodos como add(), remove() y discard() para modificar el conjunto.

En conclusión, las estructuras de datos en Python son una forma importante de organizar y manipular datos en un programa. Cada estructura tiene sus propias características y métodos útiles que pueden ser utilizados para resolver diferentes tipos de problemas. Es

importante comprender las diferencias entre las estructuras de datos para elegir la más adecuada para cada situación.