Colecciones de datos

Listas

Una lista es un tipo de datos que se utiliza para almacenar una colección ordenada de elementos. Los elementos pueden ser de diferentes tipos de datos, como números, cadenas, booleanos, etc. Las listas se definen utilizando corchetes [], y los elementos se separan por comas ,.

Aquí hay algunos ejemplos de cómo definir listas en Python:

- Lista de números enteros:

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

- Lista de cadenas:

```
nombres = ['Ana', 'Juan', 'María', 'Pedro']
```

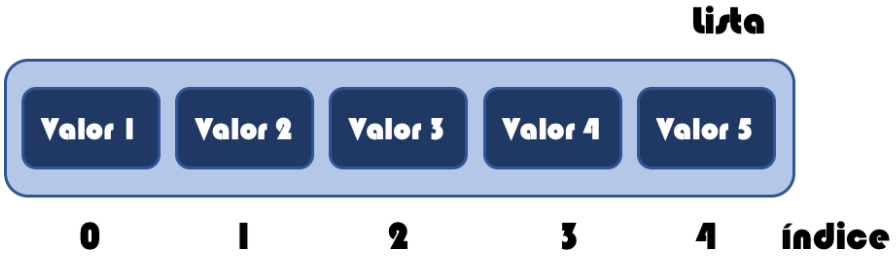
- Lista de elementos de diferentes tipos:

```
datos = [1, 'Hola', True, 3.14]
```

- Lista vacía:

```
lista_vacia = []
```

Las listas en Python son objetos iterables, lo que significa que se puede acceder a cada elemento de la lista de forma individual utilizando un índice numérico. El índice de la lista comienza en cero, lo que significa que el primer elemento de la lista tiene un índice de 0, el segundo elemento tiene un índice de 1, y así sucesivamente.



Por ejemplo, para acceder al segundo elemento de la lista nombres, puede hacer lo siguiente:

```
print(nombres[1])   # esto imprimirá 'Juan'
```

También puede modificar los elementos de una lista utilizando el índice:

```
numeros[0] = 10     # esto cambiará el primer elemento de la lista a 10
```

Se puede acceder a los elementos de una lista utilizando un índice negativo, lo que significa que comienza a contar desde el final de la lista en lugar del principio. El índice -1 hace referencia al último elemento de la lista, el índice -2 hace referencia al penúltimo elemento y así sucesivamente.

Aquí hay algunos ejemplos para ilustrar cómo funciona el acceso a elementos con índices negativos en Python:

```
numeros = [1, 2, 3, 4, 5]
print(numeros[-1])  # imprime 5, el último elemento de la lista
print(numeros[-2])  # imprime 4, el penúltimo elemento de la lista
```

También puede utilizar índices negativos para modificar elementos de la lista:

```
numeros[-1] = 10    # modifica el último elemento de la lista a 10
print(numeros)      # imprime [1, 2, 3, 4, 10]
```

Tenga en cuenta que el uso de índices negativos no cambia el orden de los elementos en la lista, sino que simplemente proporciona una forma conveniente de acceder a los elementos desde el final de la lista.

Se puede acceder a un subconjunto de elementos en una lista utilizando la sintaxis de "rebanado" (slicing). Esto le permite extraer una porción de la lista en lugar de acceder a un solo elemento. La sintaxis para rebanar una lista es [inicio:fin], donde inicio es el índice del primer elemento que se desea incluir en la rebanada y fin es el índice del primer elemento que se desea excluir de la rebanada.

Por ejemplo, suponga que tenemos la siguiente lista:

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

Podemos acceder a un subconjunto de esta lista de la siguiente manera:

```
print(numeros[1:4])   # imprime [2, 3, 4], los elementos con índices 1, 2 y 3
```

También podemos utilizar índices negativos para especificar la rebanada:

```
print(numeros[-3:-1])   # imprime [3, 4], los elementos con índices -3 y -2
```

Si omitimos el índice inicio, Python asume que queremos comenzar al principio de la lista. Si omitimos el índice fin, asume que queremos ir hasta el final de la lista. Por ejemplo:

```
print(numeros[:3])   # imprime [1, 2, 3], los primeros tres elementos
print(numeros[2:])   # imprime [3, 4, 5], todos los elementos desde el índice 2
```

También podemos especificar un tercer parámetro opcional para la sintaxis de rebanado, que es el "paso" o la cantidad de elementos que se deben saltar. Por ejemplo, si queremos seleccionar cada segundo elemento en una lista, podemos hacer lo siguiente:

```
print(numeros[::2])   # imprime [1, 3, 5], cada segundo elemento de la lista
```

En Python, es posible tener una lista dentro de otra lista, lo que se conoce como una lista anidada. Puede acceder a los elementos de la lista interna utilizando índices adicionales para navegar por las listas anidadas.

Supongamos que tenemos la siguiente lista anidada:

```
matriz = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Podemos acceder a un elemento específico en la lista interna de la siguiente manera:

```
print(matriz[1][2])   # imprime 6, el elemento en la fila 1 y columna 2
```

En este caso, matriz[1] devuelve la segunda lista anidada [4, 5, 6], y matriz[1][2] accede al tercer elemento en esa lista anidada.

Operaciones con listas

Hay muchas operaciones que se pueden realizar con listas. Aquí hay algunos ejemplos:

Agregar elementos a una lista

Para agregar un elemento al final de una lista, podemos usar el método append:

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

También podemos agregar varios elementos a una lista utilizando el método extend:

```
lista = [1, 2, 3]
lista.extend([4, 5, 6])
print(lista)   # imprime [1, 2, 3, 4, 5, 6]
```

El método insert se utiliza para insertar un elemento en una lista en un índice específico. La sintaxis es la siguiente:

```
lista.insert(indice, elemento)
```

Donde indice es el índice donde se insertará el elemento y elemento es el valor que se va a insertar.

Por ejemplo, si tenemos la siguiente lista:

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

Podemos insertar el valor 6 en el índice 2 (es decir, entre los elementos 2 y 3) de la siguiente manera:

```
lista.insert(2, 6)
```

Después de la operación de inserción, la lista se verá así:

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

El elemento 6 ha sido insertado en el índice 2 y los elementos que estaban en los índices posteriores se han desplazado hacia la derecha.

Eliminar elementos de una lista

Para eliminar un elemento de una lista por su valor, podemos utilizar el método remove:

```
lista = [1, 2, 3, 4, 5]
lista.remove(3)
print(lista)   # imprime [1, 2, 4, 5]
```

También podemos eliminar un elemento de una lista por su índice utilizando la palabra clave del:

```
lista = [1, 2, 3, 4, 5]
del lista[2]
print(lista)   # imprime [1, 2, 4, 5]
```

Concatenar listas

Podemos concatenar dos listas utilizando el operador +:

```
lista1 = [1, 2, 3]
lista2 = [4, 5, 6]
lista3 = lista1 + lista2
print(lista3)   # imprime [1, 2, 3, 4, 5, 6]
```

Ordenar una lista

Podemos ordenar una lista utilizando el método sort:

```
lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
lista.sort()
print(lista)   # imprime [1, 1, 2, 3, 3, 4, 5, 5, 5, 6, 9]
```

También podemos ordenar una lista en orden inverso utilizando el parámetro reverse:

```
lista = [3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5]
lista.sort(reverse=True)
print(lista)   # imprime [9, 6, 5, 5, 5, 4, 3, 3, 2, 1, 1]
```

Contar elementos en una lista

Podemos contar el número de veces que aparece un elemento en una lista utilizando el método count:

```
lista = [1, 2, 3, 4, 5, 3, 2, 1, 4, 3, 5]
conteo = lista.count(3)
print(conteo)   # imprime 3, ya que el valor 3 aparece tres veces en la lista
```

Tuplas

Las tuplas son una estructura de datos en Python que tienen las siguientes características:

- Son estructuras de datos inmutables: una vez que se ha creado una tupla, no se pueden modificar sus elementos. Si intentamos modificar una tupla, se generará un error.
- Se definen utilizando paréntesis y separando los elementos por comas: la sintaxis para crear una tupla es (elemento1, elemento2, ...). Si la tupla tiene un solo elemento, debemos incluir una coma después del elemento.
- Pueden contener cualquier tipo de objeto: los elementos de una tupla pueden ser cualquier tipo de objeto válido en Python, incluyendo números, cadenas de texto, listas, diccionarios, funciones, otras tuplas, etc.
- Pueden tener cualquier longitud: no hay límite en la cantidad de elementos que puede tener una tupla. Podemos crear tuplas vacías o tuplas con un solo elemento.
- Se acceden a sus elementos utilizando un índice: podemos acceder a los elementos de una tupla utilizando un índice numérico, de la misma manera que en las listas.
- Pueden ser utilizadas como claves de diccionario: como las tuplas son estructuras de datos inmutables, pueden ser utilizadas como claves de diccionario en Python. Esto es útil cuando necesitamos asociar un valor a un par de elementos que no queremos que cambien.
- Son más eficientes que las listas: como las tuplas son estructuras de datos inmutables, su tamaño no cambia durante la ejecución del programa. Esto hace que sean más eficientes en términos de memoria y

velocidad que las listas, especialmente para estructuras de datos pequeñas o medianas.

Las tuplas son una estructura de datos inmutable, es decir, una vez creada no puede ser modificada. Se definen utilizando paréntesis y separando los elementos por comas. Veamos un ejemplo:

```
tupla = (1, 2, 3, "cuatro", 5.0)
```

En este ejemplo, hemos creado una tupla llamada tupla que contiene cinco elementos: un entero (1), otro entero (2), un tercero entero (3), una cadena de texto ("cuatro") y un número de punto flotante (5.0).

Podemos acceder a los elementos de una tupla utilizando un índice, de la misma manera que en las listas:

```
print(tupla[0])  # Imprime 1
print(tupla[3])  # Imprime "cuatro"
```

Sin embargo, no podemos modificar los elementos de una tupla:

```
tupla[1] = 10  # Esto genera un error
```

También podemos crear tuplas vacías:

```
tupla_vacia = ()
```

Y tuplas con un solo elemento:

```
tupla_un_elemento = (1,)
```

Es importante tener en cuenta que cuando una tupla tiene un solo elemento, debemos incluir una coma al final, de lo contrario Python lo tratará como un valor distinto al que pretendemos.

Podemos acceder a los elementos de una tupla utilizando un índice, de la misma manera que en las listas. Los índices en Python comienzan en 0, por lo

que el primer elemento de una tupla está en el índice 0, el segundo en el índice 1, y así sucesivamente.

Veamos un ejemplo:

```
tupla = (1, 2, 3, "cuatro", 5.0)
```

Podemos acceder al primer elemento de la tupla de la siguiente manera:

```
print(tupla[0])  # Imprime 1
```

También podemos acceder al último elemento de la tupla utilizando un índice negativo:

```
print(tupla[-1])  # Imprime 5.0
```

Diccionarios

Los diccionarios son una estructura de datos que nos permite almacenar información de manera organizada utilizando una clave para acceder a cada valor. Cada clave debe ser única y puede asociarse a cualquier valor, como números, cadenas de texto, listas, tuplas, funciones, etc. Los diccionarios se definen utilizando llaves ({}) y separando cada clave y su valor con dos puntos (:).

Por ejemplo, podemos crear un diccionario que represente las edades de un grupo de personas:

```
edades = {'Juan': 25, 'María': 30, 'Pedro': 20, 'Laura': 27}
```

En este diccionario, las claves son los nombres de las personas ('Juan', 'María', 'Pedro', 'Laura') y los valores son sus edades (25, 30, 20, 27). Podemos acceder a la edad de Juan utilizando la clave 'Juan':

```
>>> edades['Juan']
25
```

También podemos agregar una nueva clave y valor al diccionario utilizando la siguiente sintaxis:

```
edades['Ana'] = 22
```

De esta manera, hemos agregado la clave 'Ana' con el valor 22 al diccionario. Podemos comprobar que se ha agregado correctamente consultando el valor asociado a la clave 'Ana':

```
>>> edades['Ana']
22
```

Además, podemos obtener una lista con todas las claves del diccionario utilizando el método keys():

```
>>> edades.keys()
dict_keys(['Juan', 'María', 'Pedro', 'Laura', 'Ana'])
```

También podemos obtener una lista con todos los valores del diccionario utilizando el método values():

```
>>> edades.values()
dict_values([25, 30, 20, 27, 22])
```

Y finalmente, podemos obtener una lista de tuplas que contienen la clave y el valor de cada elemento del diccionario utilizando el método items():

```
>>> edades.items()
dict_items([('Juan', 25), ('María', 30), ('Pedro', 20), ('Laura', 27), ('Ana', 22)])
```

En resumen, los diccionarios en Python nos permiten almacenar información organizada utilizando claves y valores. Son muy útiles para representar datos estructurados y permiten un acceso eficiente a los datos mediante las claves.