TEXT CLASS REVIEW

TEMAS A TRATAR EN EL CUE

- ¿Qué son las estructuras de datos?
- ¿Para qué se utilizan las estructuras de datos?
- Las estructuras de datos de Python: listas, tuplas, sets y diccionarios
- ¿Qué problemas se pueden resolver?
- Una estructura para cada tipo de problema
- Iterando estructuras de datos
- Crear Listas, Tuplas, Sets, Diccionarios
- Acceder a un elemento de una estructura
- Contar los elementos de una estructura
- Iterar sobre los elementos de una estructura
- Insertar y borrar elementos dentro de una estructura

¿QUÉ SON LAS ESTRUCTURAS DE DATOS?

Son la forma en que podemos almacenar y recuperar datos. Ya anteriormente hablamos de las listas, tuplas, sets y diccionarios; por lo tanto, sabes que las listas son secuenciales y que se accede a los datos por medio de un índice, mientras que los diccionarios utilizan una clave con nombre para almacenar y recuperar información.

Las estructuras de datos que existen en los lenguajes de programación son bastante similares a los sistemas del mundo real que utilizamos fuera de la esfera digital. Imagina que vas al supermercado. En este supermercado en particular, la pizza congelada se almacena junto con otros congelados, y la leche junto con otras bebidas. La tienda no tiene carteles que indiquen dónde se encuentran los distintos artículos. Así que en una tienda de comestibles desorganizada te resultaría muy difícil encontrar lo que buscas.

Afortunadamente, la mayoría de las tiendas de comestibles tienen un orden claro en la forma en que se abastecen, y en la que se distribuyen. Del mismo modo, las estructuras de datos nos

proporcionan una forma de organizar la información (incluyendo otras estructuras de datos) en un espacio digital.

¿PARA QUÉ SE UTILIZAN LAS ESTRUCTURAS DE DATOS?

Éstas se encargan de cuatro funciones principales:

- Introducir información.
- Procesar la información.
- Mantener la información.
- Recuperar información.

La entrada de información se refiere en gran medida a cómo se reciben los datos, el tipo de dato que se incluirá, cómo se almacenará éste dentro de la estructura, y cómo serán ordenados.

El procesamiento se refiere a la forma en que se manipulan los datos en la estructura de datos. Esto puede ocurrir simultáneamente, o como resultado de otros procesos que manejan las estructuras de datos. ¿Cómo tienen que cambiar los datos existentes que se han almacenado para dar cabida a datos nuevos, actualizados o eliminados?

El mantenimiento se centra en cómo se organizan los datos dentro de la estructura. ¿Qué relaciones deben mantenerse entre los datos? ¿Cuánta memoria debe reservar (asignar) el sistema para acomodar los datos?

La recuperación se dedica a encontrar y devolver los datos almacenados en la estructura. ¿Cómo podemos acceder de nuevo a esa información? ¿Qué pasos debe seguir la estructura de datos para devolvernos la información?

Los diferentes tipos y casos de uso de los datos se adaptan mejor a diferentes formas de introducción, procesamiento, almacenamiento, y recuperación; es por eso que tenemos varias estructuras de datos para elegir, las cuales están adecuadas a diferentes necesidades.

LAS ESTRUCTURAS DE DATOS DE PYTHON

Python tiene cuatro estructuras de datos no primitivas incorporadas: Listas, Diccionario, Tupla y Conjunto. Éstas cubren casi el 80% de las estructuras de datos del mundo real.

Aun cuando es posible crear nuestras propias estructuras de datos, en este módulo nos centraremos solo en estas cuatro.

Listas:

Las listas en Python son uno de los tipos de objetos de colección más versátiles disponibles. Los otros dos tipos son los diccionarios y las tuplas, pero en realidad son más bien variaciones de las listas.

Éstas hacen el trabajo de la mayoría de las estructuras de datos de colección que se encuentran en otros lenguajes. Pueden usarse para cualquier tipo de objeto, desde números y cadenas, e incluso otras listas.

Se accede a ellas igual que a las cadenas (por ejemplo, cortando y concatenando), por lo que son sencillas de usar y tienen longitud variable, es decir, crecen y se reducen automáticamente a medida que se usan.

Tuplas:

Las tuplas de Python funcionan exactamente como las listas de Python, excepto que son inmutables, es decir, no pueden ser modificadas.

Generalmente se escriben dentro de paréntesis para distinguirlas de las listas (que usan corchetes), sin embargo, los paréntesis no siempre son necesarios.

Como las tuplas son inmutables, su longitud es fija. Para aumentar o reducir una tupla, hay que crear una nueva.

Sets:

Python Set es una colección desordenada de datos que es mutable, y no permite ningún elemento duplicado. Los conjuntos se utilizan básicamente para incluir pruebas de pertenencia y eliminar entradas duplicadas.

Las llaves ({ }) representan a los sets, y los objetos colocados dentro de llaves se tratan como un set.

Diccionarios:

El diccionario de Python es como las tablas hash en cualquier otro lenguaje. Es una colección desordenada de valores de datos, usada para almacenar valores de datos como un mapa, que, a diferencia de otros Tipos de Datos que mantienen un solo valor como elemento, el Diccionario mantiene el par clave:valor.

La clave-valor se proporciona en el diccionario para hacerlo más optimizado.

La indexación del diccionario de Python se hace con la ayuda de claves. Éstas son de cualquier tipo *hashable*, es decir, un objeto que nunca puede cambiar como cadenas, números, tuplas, entre otros.

Podemos crear un diccionario usando llaves ({ }).

¿QUÉ PROBLEMAS SE PUEDEN RESOLVER?

Existen múltiples usos, y por lo tanto problemas que se pueden resolver en programación haciendo uso de las estructuras de datos. Sin embargo, a continuación, haremos una lista de las razones más comunes para utilizar las estructuras de datos. No hablaremos exclusivamente del lenguaje Python, sino de su uso en general.

- **Almacenamiento de datos**: las estructuras de datos se utilizan para la persistencia eficiente de los datos, como la especificación de la colección de atributos, y las estructuras correspondientes utilizadas para almacenar registros en un sistema de gestión de bases de datos.
- **Gestión de recursos y servicios**: los recursos y servicios básicos del sistema operativo (SO) se habilitan mediante el uso de estructuras de datos, como las listas para la asignación de memoria, la gestión de directorios de archivos, y los árboles de estructuras de archivos, así como las colas de programación de procesos.
- **Intercambio de datos**: las estructuras de datos definen la organización de la información compartida entre aplicaciones, como los paquetes TCP/IP.
- **Ordenar y clasificar**: algunas estructuras de datos proporcionan métodos eficientes de ordenación de objetos, como las cadenas de caracteres utilizadas como etiquetas. Con algunas estructuras de datos los programadores pueden gestionar elementos organizados según una prioridad específica.

- **Indexación**: las estructuras de datos más sofisticadas se utilizan para indexar objetos, como los almacenados en una base de datos.
- **Búsqueda: l**os índices creados mediante algunas estructuras aceleran la capacidad de encontrar un elemento específico buscado.
- **Escalabilidad**: las aplicaciones de big data utilizan estructuras de datos para asignar y gestionar el almacenamiento de datos en ubicaciones de almacenamiento distribuidas, lo que garantiza la escalabilidad y el rendimiento.

UNA ESTRUCTURA PARA CADA TIPO DE PROBLEMA

Dependiendo de qué problema necesitemos resolver, dependerá la estructura de datos a escoger, pues cada una está diseñada para un tipo de problema específico.

Si necesitamos llevar la cuenta de la secuencia, utilizamos una lista o tupla.

Si sólo queremos hacer seguimiento de los valores únicos y no importa el orden, utilizamos un set.

Si no necesitamos hacer cambios una vez que se ha definido tu objeto, utilizamos una tupla para ahorrar espacio y asegurarnos de que nada pueda sobrescribir los datos.

Si necesitamos seguir y modificar datos estructurados en pares clave-valor, utilizamos un diccionario.

USANDO ESTRUCTURAS PARA MODELAR PROBLEMAS COMUNES

Las estructuras que hemos mencionado hasta aquí nos permiten resolver pequeños problemas lógicos que pueden ser las piezas que necesitamos para resolver problemas aún más complejos en nuestro código. A lo largo de este CUE veremos algunos ejemplos de cómo podemos modelar problemas comunes usando estructuras.

ITERANDO ESTRUCTURAS DE DATOS

Iterar sobre una estructura de datos significa moverse a través de ella. Existen varios métodos para hacerlo, algunos de ellos comunes para varias estructuras. Más adelante estudiaremos los métodos más comunes para cada una de las 4 estructuras, los cuales incluyen: ciclos, la función `range()`, y la función `enumerate()`.

La función `range()` es útil para iterar sobre varias estructuras. Ésta genera una secuencia de enteros desde el valor inicial hasta el valor final/parada, pero no incluye el valor final en la secuencia, es decir, el número/valor de parada en la secuencia resultante.

Su sintaxis es:

```
range (comienzo, final, paso)
```

- **Comienzo** (límite superior, opcional): este parámetro se utiliza para proporcionar el valor/índice inicial de la secuencia de enteros que se va a generar. Si no se suministra, se asume por defecto que es cero.
- **Final** (límite inferior, requerido): este parámetro se utiliza para proporcionar el índice final de la secuencia de enteros a generar, es decir, cuántos elementos se van a generar.
- **Paso** (opcional): proporciona el incremento en enteros entre el número actual generado, y el siguiente.

Ejemplo:

```
vals = range(5)
print(vals)

#Resultado:
[0, 1, 2, 3, 4]

vals = range(2, 3, 3)
print(vals)

#Resultado:
[2, 5, 8]
```

Veamos también la función `enumerate()`. Ésta añade un contador a la lista, o a cualquier otro iterable, y lo devuelve como un objeto enumerado por la función.

De este modo, reduce la sobrecarga de mantener un recuento de los elementos mientras la operación de iteración.

Su sintaxis es:

```
enumerate(iterable, primer_indice)
```

Donde **iterable** es la estructura sobre la que se iterará, y **primer_indice** es el índice del elemento a partir del cual hay que registrar el contador para el iterable.

CODIFICANDO SOBRE ESTRUCTURAS

Crear Listas, Tuplas, Sets, Diccionarios

- **Listas**: para crearlas se utilizan los corchetes, y se declaran los datos o elementos correspondientes. Si no se pasan elementos a los corchetes, devuelve una lista vacía.

```
#creando una lista vacia
dias = [ ]
print(dias)

Resultado:
[ ]

#creando una lista con datos
dias = [“Lunes”, “Martes”, “Miercoles”, “Jueves”, “Viernes”, “Sabado”,
“Domingo” ]
print(dias)

#Resultado:
[“Lunes”, “Martes”, “Miercoles”, “Jueves”, “Viernes”, “Sabado”,
“Domingo” ]
```

- **Tuplas**: crearlas es tan sencillo como poner diferentes valores separados por comas. Usualmente ponemos este grupo de valores entre paréntesis para facilitar su lectura, sin embargo, esto es opcional.

```
#creando una tupla usando paréntesis
dias = (“Lunes”, “Martes”, “Miercoles”, “Jueves”, “Viernes”, “Sabado”,
“Domingo” )
print(dias)

```

```
#Resultado:
"Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado",
"Domingo"

#creando una tupla sin usar paréntesis
dias = "Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado",
"Domingo"
print(dias)

#Resultado
"Lunes", "Martes", "Miercoles", "Jueves", "Viernes", "Sabado",
"Domingo"
```

Como podemos ver, el resultado es exactamente el mismo si usamos o no los paréntesis.

- **Sets**: se crean colocando todos los elementos dentro de llaves { }, separados por comas, o utilizando la función incorporada `set()`.
  Cuando utilizamos la función `set()` todos los elementos duplicados de la lista se eliminarán automáticamente, conservando solo la primera aparición de cada elemento.

```
mi_set = {1, 2, 3}
print(mi_set)

#Resultado:
{1, 2, 3}

#creando un set a partir de una lista con elementos duplicados
mi_lista = [1, 2, 3, 4, 2, 3]
mi_set = set(mi_lista)
print(mi_set)

#Resultado
{1, 2, 3, 4}

#creando un set con datos de diferentes tipos
mi_set = {1.2, "hola", (1, 2, 3)}
print(mi_set)

#Resultado
 {1.2, "hola", (1, 2, 3)}
```

- **Diccionarios**: crearlos es tan sencillo como colocar los elementos dentro de llaves { } separadas por comas. Cada elemento tiene una clave, y un valor correspondiente que se expresa como un par (clave: valor).

También podemos crear un diccionario haciendo uso de la función `dict()`

```
#creando un diccionario vacío
mi_diccionario = { }

# creando un diccionario con claves mixtas
mi_diccionario = {'nombre': 'Juan', 1: [2, 4, 3]}

# creando un diccionario usando dict()
mi_diccionario = dict({1:'manzana', 2:'pelota'})

"""creando un diccionario a partir de la secuencia que tiene cada
elemento como un par"""
mi_diccionario = dict([(1,'manzana'), (2,'pelota')])
```

ACCEDER A UN ELEMENTO DE UNA ESTRUCTURA

Cada una de las estructuras tiene sus propios métodos de acceso a los datos, algunos son comunes a varias de ellas. Revisemos estos métodos en detalle.

- **Listas y tuplas**: el método para acceder a los datos en las listas y las tuplas es a través de su índice, para ello simplemente debemos hacer uso del operador corchete, el cual nos dará acceso inmediato al dato según el índice que indiquemos.
  Su sintaxis es:

```
lista[indice]
```

  También haciendo uso del operador corchetes podemos obtener un grupo de valores. Para ello debemos indicar el índice inicial y el final separado de los dos puntos (:)
  Sintaxis:

```
lista[índice_inferior : índice_superior]
```

  Si el índice inferior o el superior no son indicados, automáticamente se supondrá el índice cero en caso de omitirse el inferior, o el mayor índice encontrado en la lista, en caso de omitirse el superior.
- **Sets**: no se puede acceder a los elementos del set haciendo referencia a un índice, ya que los sets no están ordenados y los elementos no tienen índice. Sin embargo, se puede recorrer los elementos del set utilizando un ciclo **for**, o preguntar si un valor específico está presente

en un conjunto, utilizando la palabra clave in; sin embargo, el resultado de la búsqueda será verdadero (true) o falso (false).

- **Diccionarios**: para acceder a los elementos del diccionario se puede utilizar también el operador corchete, junto con la clave para obtener su valor.

CONTAR LOS ELEMENTOS DE UNA ESTRUCTURA

Para conocer el número de elementos que tiene cada estructura lo hacemos usando la función `len()`, la cual devuelve un entero con el número total de elementos dentro de ella.

La sintaxis es:

```
1 len(estructura)
```

Esta función funciona directamente con las listas, tuplas y sets. Sin embargo, cuando se trata de los diccionarios, debemos dar un paso adicional.

Para saber la cantidad de elementos de un diccionario también utilizaremos la función `len()`, pero debemos hacerlo en combinación con la función `keys()`. La función `keys()` nos regresará una lista con todas las claves del diccionario, y esta lista es la que contaremos.

ITERAR SOBRE LOS ELEMENTOS DE UNA ESTRUCTURA

Lista o tupla

Recordemos que las listas y las tuplas son prácticamente iguales, lo que las diferencia es que las tuplas no pueden modificarse. Para iterar sobre una tupla no debemos modificarla, por lo tanto, los métodos de iteración son idénticos para ambas estructuras.

Para el caso de los ejemplos usaremos listas, pero debemos recordar que el método de iteración también funcionará en las tuplas.

1) Usando la función `range()`:
   Para ello chequeamos la cantidad de datos que contiene la lista con la función `len()`, y con ese valor crearemos el rango. Luego, dentro del cuerpo accederemos a cada dato de la lista con el operador corchetes.

Ejemplo:

```
lista = [10, 50, 75, 83]

for x in range(len(lista)):
    print(lista[x])
#Resultado:
10
50
75
83
```

2) Usando un ciclo **for**:

Ejemplo:

```
lista = [10, 50, 75, 83]

for x in lista:
    print(x)
#Resultado:
10
50
75
83
```

3) Usando un ciclo **while:**

Para ello creamos un contador, el cual se actualizará al finalizar cada iteración. La condición del **while** es que el contador sea menor que la cantidad de elementos que tiene la lista, y para ello usamos la función `len()`. Recordemos que el primer índice de la lista es cero. Luego accedemos a cada elemento usando el operador corchetes y el contador.

```
lista = [10, 50, 75, 83]
while x < len(lista):
    print(lista[x])
    x = x+1
#Resultado:
10
50
75
83
```

4) Usando la función `enumerate()`:
   La función `enumerate()` nos devuelve en cada iteración la clave y el valor de cada elemento, y debemos asignar cada uno a una variable en el ciclo **for**. Luego simplemente utilizamos el valor en cada iteración.

```
lista = [10, 50, 75, 83]

for x, val in enumerate(lista):
        print (val)

#Resultado:
10
50
75
83
```

Sets

Los sets son colecciones desordenadas, por lo que no se puede acceder a los elementos mediante la indexación. Si queremos acceder a los elementos del conjunto, tendremos que iterar con la ayuda de sentencias de bucle.
Algunas de las formas de iterar a través de conjuntos en Python son:

1) Usando el ciclo **for:**
   La solución es la misma que en las listas.
2) Usando la function `enumerate()`**:**
   La solución es la misma que en las listas.
3) Usando la function `iter()`**:**
   Se recorre el set con un ciclo **for**, el cual realmente iterará sobre los resultados de la función `iter()`. Cada elemento de `iter()` se debe almacenar en una variable, que se actualizará en cada ciclo.

```
mi_set = {'london', 'new york', 'seattle', 'sydney'}

for item in iter(mi_set):
    print(item)

#Resultado:
```

```
new york
seattle
london
Sydney
```

4) Convertir el set en una lista:

Para ello simplemente usamos la función **`list()`**, a la cual pasamos el set como parámetro.

```
mi_lista = list(mi_set)
```

Y ahora usamos cualquiera de las estrategias vistas en el ejemplo anterior para recorrer **`mi_lista`**.

Diccionarios

El diccionario en Python es una colección desordenada de datos, utilizada para almacenar valores de datos como un mapa. A diferencia de otros tipos de datos que mantienen un solo valor como elemento, el diccionario mantiene el par clave: valor.

Para iterar sobre un diccionario podemos utilizar los siguientes métodos:

1) Usando la clave de acceso mediante la función **`.keys()`**:

Esta función nos permite obtener todas las claves del diccionario. Luego usamos un ciclo **for** para recorrer el resultado de **`keys()`** que debimos almacenar en una variable, y en cada iteración utilizaremos la clave actual para acceder al valor del par con el operador corchetes.

Ejemplo:

```
paises_y_capitales = {'Japon': 'Tokio', 'Inglaterra': 'Londres',
'Francia': 'Paris', 'Alemania': 'Berlin'}

keys = paises_y_capitales.keys()

for clave in keys :
    print(f"{clave} - {paises_y_capitales[clave]}")
#Resultado:
Japon - Tokio
Inglaterra - Londres
Francia - Paris
Alemania - Berlin
```

2) Usando el ciclo **for** para iterar sobre las claves:

Al iterar sobre el diccionario de esta forma tendremos acceso a las claves de cada par clave:valor. Así que luego, usando el operador corchetes, podemos acceder a cada valor del par.

```
paises_y_capitales = {'Japon': 'Tokio', 'Inglaterra': 'Londres',
'Francia': 'Paris', 'Alemania': 'Berlin'}

for pais in paises_y_capitales:
    print(f"{pais} - {paises_y_capitales[pais]}")

#Resultado:
Japon - Tokio
Inglaterra - Londres
Francia - Paris
Alemania - Berlin
```

3) Usando el ciclo **for** y la función **`values()`**:

Esta función nos devuelve solo los valores del par clave:valor del diccionario.

```
paises_y_capitales = {'Japon': 'Tokio', 'Inglaterra': 'Londres',
'Francia': 'Paris', 'Alemania': 'Berlin'}

for capital in paises_y_capitales.values():
    print(capital)


#Resultado:
Tokio
Londres
Paris
Berlin
```

4) Usando el ciclo **for** y la función **`ítems()`**:

Esta función nos regresa el par clave:valor de cada elemento del diccionario. Haremos uso del ciclo **for** para iterar sobre los valores que nos regresa la función.

```
paises_y_capitales = {'Japon': 'Tokio', 'Inglaterra': 'Londres',
'Francia': 'Paris', 'Alemania': 'Berlin'}
```

```
 3
 4 for pais, capital in paises_y_capitales.items():
 5     print(pais, "-", capital)
 6
 7 #Resultado:
 8 Japon - Tokio
 9 Inglaterra - Londres
10 Francia - Paris
11 Alemania - Berlin
```

INSERTAR Y BORRAR ELEMENTOS DENTRO DE UNA ESTRUCTURA

Listas

Para agregar elementos a una lista en Python existen varios métodos:

- **Usando la función append():** se pueden añadir elementos a la lista utilizando la función incorporada `append()`. Sólo se puede añadir un elemento a la vez, para la adición de múltiples elementos se utilizan ciclos.

  Sintaxis:

```
1 lista.append(elemento_a_agregar)
```

  Ejemplo:

```
 1 mi_lista = [ ]
 2 mi_lista.append(1)
 3 mi_lista.append(2)
 4 mi_lista.append(3)
 5 print(mi_lista)
 6
 7 #Resultado:
 8 [1, 2, 3]
 9
10 #usando un ciclo
11 for i in range(4, 6)
12     mi_lista.append(i)
13
14 print(mi_lista)
15
16 #Resultado
17 [1, 2, 3, 4, 5]
```

- **Usando la función insert():** el método `append()` sólo funciona para añadir elementos al final de la Lista, y para añadir elementos en la posición deseada se utiliza el método `insert()`. A diferencia de `append()` que sólo toma un argumento, el método `insert()` requiere dos argumentos (posición y valor).

  Recordemos que las listas comienzan con el índice cero, por lo tanto, si definimos el argumento posición con el valor cero, estaríamos posicionando el dato en el comienzo de la lista.

  Sintaxis:

```
lista.insert(posicion, valor)
```

  Ejemplo:

```
mi_lista = [1, 2, 3, 4]
mi_lista.insert(0, "Comienzo")
mi_lista.insert(4, "nuevo dato")
print(mi_lista)

#Resultado:
["Comienzo", 1, 2, 3, "nuevo dato", 4]
```

- **Usando la función extend():** este método se utiliza para añadir múltiples elementos al mismo tiempo al final de la lista.

  Sintaxis:

```
lista.extend(lista_a_agregar)
```

  Ejemplo:

```
mi_lista = [1, 2, 3, 4]
mi_lista.extend([10, 20, 30])
print(mi_lista)

#Resultado:
[1, 2, 3, 4, 10, 20, 30]
```

  También podemos eliminar elementos de una lista, para ellos tenemos varias opciones.

- **Usando la función remove():** los elementos pueden ser eliminados de la lista utilizando la función incorporada `remove()`, pero se produce un error si el elemento no existe en la lista.

El método `remove()` sólo elimina un elemento a la vez, por lo que para eliminar un rango de elementos se utiliza el iterador. El método `remove()` elimina el elemento especificado.

Debemos tomar en cuenta que la función `remove()` elimina solo la primera aparición de un elemento, así que si hay duplicados, debe ser eliminado en otra iteración.

Sintaxis:

```
lista.remove(valor)
```

Ejemplo:

```
mi_lista = [1, 2, 3, 4, 5, 5, 6, 7]
mi_lista.remove(2)
mi_lista.remove(5)
print(mi_lista)

#Resultado:
[1, 3, 4, 5, 6, 7]

#utilizando un ciclo

for i in range(3, 5):
    mi_lista.remove(i)

print(mi_lista)

#Resultado:
[1, 6, 7]
```

- **Usando la función pop():** también se puede utilizar para eliminar y devolver un elemento de la lista, pero por defecto sólo elimina el último elemento de la lista en caso de no pasar ningún argumento. Para eliminar un elemento de una posición específica de la Lista, se pasa el índice del elemento como argumento al método `pop()`.

  Sintaxis:

```
lista.pop(indice)
```

Ejemplo:

```
mi_lista = [1, 2, 3, 4, 5, 6]
mi_lista.pop()
elem = mi_lista.pop(1)
print(mi_lista)
```

```
print(elem)

#Resultado:
[1, 3, 4, 5]
2
```

Como se mencionó antes, la función no solo elimina un dato de la lista, sino que lo devuelve, por lo tanto, podemos asignárselo a una variable para luego usarlo, como fue el caso del ejemplo.

Tuplas

Recordemos que por definición, las tuplas son inmutables, es decir que una vez creadas no se pueden modificar, lo que implica que no podemos agregar o quitar elementos de ellas.

En caso de ser necesario, la forma de hacerlo sería crear una nueva tupla con las modificaciones. También es posible convertir la tupla en una lista, hacer las modificaciones pertinentes, y luego convertir esa lista en una tupla.

Sets

Para agregar datos a un set contamos con varias opciones:

- **Usando la función add():** los elementos pueden ser añadidos al Conjunto utilizando la función incorporada `add()`. Sólo se puede añadir un elemento a la vez al conjunto utilizando el método `add()`, los bucles se utilizan para añadir múltiples elementos a la vez con el uso del método `add()`.

  Sintaxis:

```
set.add(valor)
```

Ejemplo:

```
mi_set = set()
mi_set.add(1)
mi_set.add(5)
print(mi_set)

#Resultado:
{1, 5}
```

- **Usando la función update():** para la adición de dos o más elementos, se utiliza el método `Update()`. Éste acepta como argumentos listas, cadenas, tuplas, y otros conjuntos. En todos estos casos, se evitan los elementos duplicados.

  Sintaxis:

```
set.update(dato)
```

  Ejemplo:

```
mi_set = set([1, 2, 3, 4])
mi_set.update([5, 6])
print(mi_set)

#Resultado:
{1, 2, 3, 4, 5, 6}
```

Para eliminar elementos de un set podemos usar las siguientes opciones:

- **Usando las funciones remove() y discard():** los elementos pueden ser eliminados del conjunto utilizando la función `remove()`. Esta produce un `KeyError` si el elemento no existe en el conjunto.

  Para eliminar elementos de un conjunto sin `KeyError`, utilice `discard()`, si el elemento no existe en el conjunto, permanece sin cambios.

  Sintaxis:

```
set.remove(valor)

set.discard(valor)
```

  Ejemplo:

```
mi_set = set([1, 2, 3, 4, 5, 6 ])
mi_set.remove(2)
mi_set.discard(5)

print(mi_set)

#Resultado:
{1, 3, 4, 6}
```

- **Usando la función pop():** esta puede utilizarse para eliminar y devolver un elemento de un set, sin embargo, sólo elimina el último elemento del set.

Sintaxis:

```
set.pop()
```

Ejemplo:

```
mi_set = set([1, 4, 2, 8, 3])
mi_set.pop()

print(mi_set)

#Resultado:
{1, 4, 2, 8}
```

- **Usando la función clear():** para eliminar todos los elementos del set, se utiliza la función `clear()`.

Sintaxis:

```
set.clear()
```

Ejemplo:

```
mi_set = set([1, 2, 3, 4])
mi_set.clear()

print(mi_set)

#Resultado:
set()
```

Diccionarios

A diferencia de las listas y las tuplas, no hay ningún método `add()`, `insert()` o `append()` que podamos utilizar para añadir elementos a la estructura de datos. En su lugar, se debe crear una nueva clave de índice, que se utilizará para almacenar el valor que desea guardar en su diccionario.

Sintaxis:

```
diccionario[clave] = valor
```

Ejemplo:

```
datos_personales = {"Nombre": "Antonio", "Apellido":"Lopez"}
datos_personales["Edad"] = 30

print(datos_personales

#Resultado:
{"Nombre": "Antonio", "Apellido":"Lopez", "Edad":30}
```

Para eliminar datos de un diccionario contamos con las siguientes opciones:

- **Usando la función del**: la función elimina el elemento con la clave especificada.

  Sintaxis:

```
del diccionario[clave]
```

  Ejemplo:

```
mi_diccionario = {"Marca":"Ford", "Modelo":"Mustang", "Año":1964}
del mi_diccionario["Modelo"]

print(mi_diccionario)

#Resultado:
 {"Marca":"Ford", "Año":1964}
```

- **Usando la función clear():** para eliminar todos los elementos del diccionario, se utiliza la función `clear()`.

  Sintaxis:

```
diccionario.clear()
```

Ejemplo:

```
mi_diccionario = {"Marca":"Ford", "Modelo":"Mustang", "Año":1964}
mi_ diccionario.clear()

print(mi_ diccionario)

#Resultado:
{}
```