**Estructuras Asociativas:**

**A las estructuras asociativas se les llama de diversas formas, mapas de hash, diccionarios, tablas de hash, etc.  
Muchos lenguajes de programación incluyen esta estructura en su API  
Es una estructura muy eficiente, el acceso a los elementos es en la mayoría de los casos aleatorio  
Es una de las estructuras más poderosas y útiles  
Son estructuras indexadas, pero a diferencia de una lista, el índice puede ser de cualquier tipo no mutable  
Los elementos de una estructura asociativa se componen en dos partes:  
Llave: es el índice con el que se puede recuperar el valor  
Valor: Es el valor en si que se desea almacenar  
La idea es que a partir de una llave dada pueda almacenarse o recuperarse un valor de forma aleatoria, igual que un arreglo**

**Las estructuras asociativas estándar son los diccionarios**

**Diccionario = {‘llave1:’valor1’,’llave2’:’valor2}  
print(diccionario[‘llave1]) #imprime valor1  
  
diccionario[‘llave3’] = ‘valor3’  
print(diccionario[‘llave3]) #imprime valor3**

**Son estructuras mutables y no lineales  
La llave solo puede ser de un tipo no mutable como cadenas, tuplas, enteros, etc., asimismo, el valor de la llave puede ser el que sea**

**d1 = {3: 'tres'} # OK  
d2 = {(1,): 'uno'} # OK  
d3 = {[1, 2]: 'dos'} #error, la llave es list**

**Considerar que en estas estructuras los elementos no se ordenan de manera particular (como si pasa en las listas), por lo que no se puede predecir su orden interno o intentar ordenar (mediante sorted por ejemplo), Además, si se imprime un diccionario, es posible que los elementos no aparezcan en el orden en el que los insertaste**

**Crear diccionario vacío  
d1 = {} # forma literal  
d2 = dict() # usando función de conversión  
o también  
print(type(d1))  
print(type(d2))**

**Longitud de diccionario  
Se refiere a cuántos pares llave-valor tiene  
Como en otras estructuras se utiliza la función len  
d = {'a': 1, 'b': 2, 'c': 3}  
print(len(d))  
3**

**Agregar nueva llave-valor  
d1 = {'a': 1, 'b': 2}  
d1['c'] = 3  
print(d1)  
{'a': 1, 'b': 2, 'c': 3}**

**Cambiar el valor asociado a una llave existente  
Es exactamente igual que agregar una nueva llave-valor  
d1 = {'a': 1, 'b': 2}  
d1['a'] = 66  
print(d1)  
{'a': 66, 'b': 2}**

**Recuperar valor a partir de llave  
Hay dos formas comunes:  
Directa usando []  
Indirecta a partir de función get  
La forma directa tiene la desventaja de que si la llave no existe se lanza una excepción  
Mediante el método get se puede pasar un parámetro opcional que establece el valor por defecto a regresar si la llave no existe. En general siempre es mejor recuperar con método get  
d = {'a': 1, 'b': 2, 'c': 3}  
print(d['a']) # forma directa  
print(d.get('a')) # indirecta  
print(d.get('d', -1)) # valor por defecto  
print(d.get('b', -1)) # si existe no pasa nada  
1  
1  
-1  
2**

**Regresar llaves  
Regresa todas las llaves en una estructura similar a una lista (dict\_keys)  
De ser necesario se puede convertir el resultado a una lista normal mediante list  
Se usa el método keys  
Útil para recorrer el diccionario (visto más adelante)  
d = {'a': 1, 'b': 2, 'c': 3}  
print(d.keys())  
print(list(d.keys()))  
dict\_keys(['a', 'b', 'c'])  
['a', 'b', 'c']**

**Recorrer diccionario  
Coma tal no se puede hacer directamente dado que la estructura no es lineal  
Una forma es obteniendo primero la lista de llaves  
d = {'a': 1, 'b': 2, 'c': 3}  
for llave in d.keys():  
print(d[llave]) # no hay riesgo de que no exista  
1  
2  
3  
Se puede también recorrer tanto llaves y valores a la vez mediante el método ítems  
Dicho método regresa una lista de tuplas con los pares llave-valor  
d = {'a': 1, 'b': 2, 'c': 3}  
print(d.items())  
for k, v in d.items():  
print('llave %s, valor %s' % (k, v))  
dict\_items([('a', 1), ('b', 2), ('c', 3)])  
llave a, valor 1  
llave b, valor 2  
llave c, valor 3**

**Borrar un par llave-valor  
Se puede como en otros casos con la función del  
En general no es tan común o necesario borrar cosas  
d = {'a': 1, 'b': 2, 'c': 3}  
del(d['a'])  
print(d)  
{'b': 2, 'c': 3}**

**Determinar si una llave pertenece al diccionario  
Se puede usar el operador in**

**D = {‘a’: 1, ‘b’: 2, ‘c’: 3}  
llave = ‘b’**

**Hash**

**Un hash map (Diccionario) se implementa tradicionalmente a partir de un arreglo interno (que puede ser de dos dimensiones, a lo que se llama una tabla)  
Es en el arreglo interno donde en realidad se guardan los elementos, el arreglo interno permite que el acceso a los elementos sea aleatorio, la idea es convertir la llave de un elemento a un índice numero del arreglo interno, esta conversion se logra mediante una función hash, con el índice numerico se recupera directamente el valor**

**Funciones de hash**

**Es una función que recibe algún tipo de objeto (para nuestro caso, el tipo de la llave) y regresa un valor en un rango predefinido (normalmente un numero)**

**Si el rango es de 1000, el hash regresara algún valor entre 0 y 999, por ejemplo.**

**Por ejemplo, imaginemos una función de hash que recibe una cadena y lo que hace es devolver un valor entre 0 y 99.**

**La misma cadena siempre generara el mismo valor**

**Cualquier cadena que le pasemos sin importar su extensión generara un numero entre 0 y 99**

**Como el dominio de la función es un conjunto infinito pero el codominio es un conjunto finito es obvio que varios (de hecho un numero infinito) de los valores del dominio se mapearan con el mismo valor del codominio. A esto se le llama “Colisión”**

**Entre mas grande sea la aridad(longitud) del codominio menor será la probabilidad de que dos elementos colisionen entre si**

**def par(n): return n % 2  
par(1111) #imprime 1  
par(222) #imprime 0  
par(2221) #imprime 1 y causa colisión al dar el mismo tipo de hash**

**En una estructura asociativa debe definirse:  
 Una función hash para convertir de llave a índice, una política de manejo de colisiones  
Una política común es encadenamiento, esto es, se usa una lista ligada de los valores que colisionaron en la misma posición (de esta forma el arreglo interno guarda realmente listas ligadas)**

**Existen diversas funciones de hash con diversos propósitos, en este curso nos concentraremos en funciones de has que sean capaces de generan un numero entre 0 y el numero total -1 de elementos de nuestro arreglo interno.**

**La función debería de dispersar de la forma más equitativa posible los valores generados de tal manera que se minimicen las colisiones**

**Ejemplos de funciones:  
 Modulo: se recomienda que el modulo sea un numero primo (Dispersión más equitativa)**

**Def hashear(llave: str, modulo:int) -> int:**

**“””  
regresa el hash de una llave de tipo cadena para el valor del modulo dado  
llave : str  
modulo: int  
returns: int, hash calculado  
“””  
suma = 0  
for c in llave:  
 suma += ord( c )  
return suma % modulo**

**Modulo = 1013 #numero primo  
print(hashear(“hola mundo”, modulo)) #imprime 999**

**Dato extra: calcular mods**

**Mod 4: Si se quiere contar el mod 4 de 7, solo se hace:**

**0 (1) (5)**

**1 (2) (6)**

**2 (3) (7)**

**3 (4)**

**Funcion de horner:  
 Es una función que el modulo, aplicable normalmente cadenas, utiliza descomposición de polinomios**

**Funcion hash:**

**Python utiliza la función hash para determinar el valor numero de las llaves  
Dependiendo del tipo de llave, Python usara diferentes algoritmos de hash, Cabe mencionar que solo los tipos inmutables son hasehables**