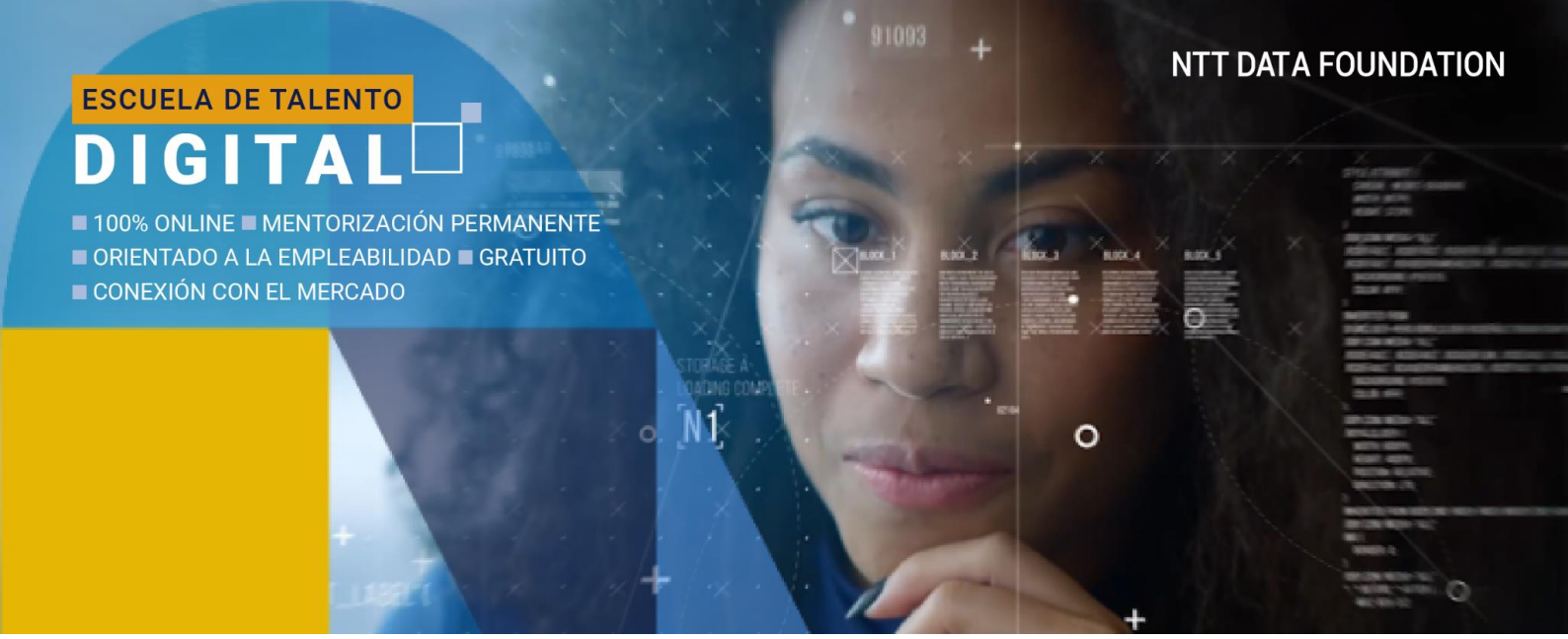

ESCUELA DE TALENTO DIGITAL

NTT DATA FOUNDATION

LISTAS, TUPLAS, SETS Y DICCIONARIOS

ÍNDICE

1. ¿QUÉ SON?

Tipo de datos que contienen conjuntos de elementos relacionados entre sí, sean del mismo tipo o no.

Se crean incluyendo los entre corchetes.

EJEMPLOS

animal = [“perro”, “gato”, “pájaro”]
numero = [1, 3, 5, 7, 9]
mezcla = [“perro”, 1, “gato”, 5, “animal”,3]

OPERACIONES

ACCESO A ELEMENTOS

Con el nombre de la lista y el índice donde se encuentra el elemento.
nombre lista[número elemento]

Recuerda que el primer índice es 0.

EJEMPLOS

animal[0] será "perro"
numero[2] sería 5
mezcla[5] = mezcla[-1] sería 3

SUSTITUCIÓN DE ELEMENTOS

Con el nombre de la lista, el índice de la posición del elemento, igual y el nuevo elemento.

EJEMPLOS

animal[2] = "rata"
animal = ["perro", "gato", "rata"]

numero[0] = 2
numero = [2, 3, 5, 7, 9]

Para utilizar y trabajar con listas tenemos una serie de métodos ya creados con una serie de funcionalidades, que nos van a permitir, entre otras cosas, añadir elementos, eliminarlos, ordenarlos, copiarlos, etc. Vamos a verlos en detalle.

2. AÑADIR ELEMENTOS

Para añadir elementos a una lista utilizamos el método `append()`. Este método se utiliza de la siguiente manera:

```
Nombre de lista.append(elemento a añadir)
```

Este método siempre añade los elementos al final de la lista. Por ejemplo, si tenemos la lista `asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”]` y queremos añadir el elemento “Lengua” a esta lista, lo haremos así:

```
asignaturas.append(“Lengua”)
```

Con lo que nuestra lista asignaturas será ahora `[“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]`

Si los elementos de nuestra lista son de algún tipo numérico y no string, sería igual, si tenemos la lista `numeros = [4, 2, 8, 6]` y queremos añadirle el número 10, haríamos lo siguiente:

```
numeros.append(10)
```

Con lo que nuestra lista `numeros` ahora sería `[4, 2, 8, 6, 10]`

3. ELIMINAR ELEMENTOS

3.1. Remove

Para eliminar elementos de una lista utilizamos el método `remove()`. Este método se utilizaría de forma similar al método `append`, concretamente:

```
Nombre de la lista.remove(elemento que queremos eliminar)
```

Por ejemplo, si tenemos nuestra lista anterior, `asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]`, y queremos eliminar el elemento “Ciencias Sociales”, tendríamos que hacerlo así:

```
asignaturas.remove(“Ciencias Sociales”)
```

Con lo que ahora nuestra lista asignaturas sería `[“Matemáticas”, “Ciencias Naturales”, “Geografía”, “Lengua”]`

Ojo que, si tenemos elementos repetidos en la lista, `remove` solo elimina el primero de los elementos que se encuentra. Por ejemplo, si tenemos nuestra lista `numeros = [4, 2, 4, 8, 6, 10]`, la sentencia `numeros.remove(4)` nos dejará la lista números con los siguientes elementos `[2, 4, 8, 6, 10]` porque nos habrá eliminado el primer 4 que se ha encontrado, pero no todos los 4 que hay en la lista.

Hay una función que permite eliminar todos los elementos duplicados de una lista, esta función tiene la siguiente estructura:

```
nombre de la lista = list(dict.fromkeys(nombre de la lista))
```

Es una función que se aplica directamente sobre la variable, por lo que no olvidéis la asignación con el igual.

Por ejemplo, si a nuestra lista `numeros = [4, 2, 8, 4, 8, 6, 10]` le aplicamos esta función de esta forma:

```
numeros = list(dict.fromkeys(numeros))
```

Obtendremos que los valores que hay ahora en `numeros` son:

```
[4, 2, 8, 6, 10]
```

3.2. Del

Para eliminar elementos de una lista a partir de su índice, podemos utilizar la función `del`. Este método se utilizaría así:

```
del nombre de la lista[índice o rango de elementos a eliminar]
```

Por ejemplo, si tenemos nuestra lista anterior, `asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]` y queremos eliminar el elemento con índice 2, nuestra sentencia sería:

```
del asignaturas[2]
```

Con lo que ahora nuestra lista sería:

```
asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Geografía”, “Lengua”]
```

Esta función también nos permite borrar un rango de elementos. Por ejemplo, si en nuestra lista `asignaturas` completa, `["Matemáticas", "Ciencias Sociales", "Ciencias Naturales", "Geografía", "Lengua"]`, queremos borrar los elementos desde el índice 1 al 3, deberíamos utilizar la siguiente sentencia:

```
del asignaturas[1:4]
```

Porque recordad que, Python, no opera en el valor del rango indicado como final, así que, si queremos borrar también el elemento de índice 3, deberemos decirle que borre hasta el 4.

Esto nos dejará la lista así:

```
['Matemáticas', 'Lengua']
```

Del también nos permite borrar desde cualquier posición del índice hasta el final. Para ello no indicaremos un valor de índice final. Por ejemplo, si en la lista completa de asignaturas, `["Matemáticas", "Ciencias Sociales", "Ciencias Naturales", "Geografía", "Lengua"]`, queremos borrar desde el elemento de índice 2 hasta el final, lo haremos con la siguiente sentencia:

```
del asignaturas[2:]
```

Lo que nos dejará la lista:

```
['Matemáticas', 'Ciencias Sociales']
```

3.3. Pop

`Pop`, al igual que `del`, elimina un elemento de una lista a partir de su índice. La diferencia es que pop no elimina rangos.

Su estructura es:

```
Nombre de la lista.pop(índice del elemento)
```

Por ejemplo, en nuestra lista `asignaturas = ["Matemáticas", "Ciencias Sociales", "Ciencias Naturales", "Geografía", "Lengua"]`, si queremos eliminar el elemento con índice 0, lo haremos así:

```
asignaturas.pop(0)
```

Con lo que nuestra lista ahora será:

```
['Ciencias Sociales', 'Ciencias Naturales', 'Geografía', 'Lengua']
```

Si en la función pop no indicamos ningún parámetro, es decir, si la llamamos, por ejemplo, como `asignaturas.pop()` eliminará el último elemento de la lista, quedando nuestra lista asignaturas así:

```
['Ciencias Sociales', 'Ciencias Naturales', 'Geografía']
```

4. COPIAR LISTAS

Cuando queremos copiar una lista en otra lista utilizamos el método `copy()`. A este método no es necesario incluirle ningún parámetro y funciona así:

```
Nombre lista copiada = lista que queremos copiar.copy()
```

Por ejemplo, si tenemos nuestra lista de números, `numeros = [4, 2, 8, 6, 10]`, y queremos una nueva lista llamada `copia_numeros`, haríamos lo siguiente:

```
copia_numeros = numeros.copy()
```

Con esto ahora tendríamos dos listas iguales:

```
numeros = [4, 2, 8, 6, 10]

copia_numeros = [4, 2, 8, 6, 10]
```

Esto es muy útil porque, cuando trabajamos con listas, si hacemos una asignación directa, por ejemplo, `copia_numeros = numeros`, lo que estamos diciendo es que esas dos listas siempre van a ser iguales, y si, después de asignarlas, hacemos un cambio en la lista

`numeros`, ese cambio también se va a realizar en la lista `copia_numeros` aunque no lo especifiquemos.

Por ejemplo, imagina que quieres tener dos listas de asignaturas, una en la que tengas las asignaturas del curso, podría servir para este ejemplo nuestra lista `asignaturas` anterior, y otra en la que tengas solo las asignaturas que te quedan por aprobar, pero cada vez que apruebas una asignatura querrás quitarla de esa segunda lista porque ya no está por aprobar.

Imagina que está empezando el curso y aún no te has examinado de ninguna asignatura, por lo que ambas listas serán iguales.

```
asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]

asignaturas_por_aprobar = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]
```

Si para crear la lista `asignaturas_por_aprobar` hacemos esto:

```
asignaturas_por_aprobar = asignaturas
```

Es verdad que ambas serán iguales y tenemos lo que queríamos, pero ya ha pasado el primer trimestre, te has examinado de Lengua y de Ciencias Naturales y quieres sacarlas de la lista `asignaturas_por_aprobar` porque has conseguido aprobarlas, si haces esto:

```
asignaturas_por_aprobar.remove(“Lengua”)

asignaturas_por_aprobar.remove(“Ciencias Naturales”)
```

Ahora, la lista `asignaturas_por_aprobar` será `[“Matemáticas”, “Ciencias Sociales”, “Geografía”]`, pero, la lista asignaturas también habrá cambiado y no contendrá todas las asignaturas, porque siempre va a ser igual a `asignaturas_por_aprobar`.

Para hacerlo bien, tendrías que hacer esto:

```
asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]

asignaturas_por_aprobar = asignaturas.copy()

asignaturas_por_aprobar.remove(“Lengua”)

asignaturas_por_aprobar.remove(“Ciencias Naturales”)
```

De esta forma, las dos listas son independientes y al eliminar los elementos “Lengua” y “Ciencias Naturales” de la lista `asignaturas_por_aprobar`, no se eliminarán de la lista asignaturas, porque ambas serán independientes. Ambas listas contendrán ahora:

`asignaturas_por_aprobar` contendrá `[“Matemáticas”, “Ciencias Sociales”, “Geografía”]`

`asignaturas` contendrá `[“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]`

5. REPETICIONES DE UN ELEMENTO

Para saber cuántas veces está repetido un elemento en una lista utilizamos el método `count()`. Este método funciona de la siguiente manera:

```
Nombre de la lista.count(elemento que queremos saber cuántas veces está en la lista)
```

Por ejemplo, si en nuestra lista de números anterior, `numeros = [4, 2, 4, 8, 6, 10]` queremos saber cuántas veces aparece el número 4, tendremos la siguiente línea de código, `numeros.count(4)`, que nos devolverá como resultado `2`, porque 4 está dos veces en la lista. Sin embargo, la sentencia `numeros.count(10)` nos devolverá como resultado `1`, porque 10 está solo una vez en la lista.

6. UNIR DOS LISTAS

Cuando queremos añadir los elementos de una lista a otra lista, utilizamos el método extend(). Funciona de la siguiente manera:

Nombre de la lista a la que queremos añadir elementos.extend(nombre de la lista que contiene los elementos que queremos añadir)

Por ejemplo, si tenemos las listas:

asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]

numeros = [4, 2, 4, 8, 6, 10]

La sentencia numeros.extend(asignaturas) mantendrá la lista asignaturas sin variar, pero la lista numeros ahora será esta:

numeros = [4, 2, 4, 8, 6, 10, “Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]

Como veis, los elementos que se añaden se sitúan después de los que ya están en la lista.

7. POSICIÓN DE UN ELEMENTO

El método `index()` nos devuelve la posición, o el valor del índice, de la primera vez que se encuentra un elemento especificado. Esto funciona de la siguiente manera:

```
Nombre de la lista.index(elemento del que queremos conocer su posición)
```

Ten en cuenta que la posición, o valor del índice, siempre empieza en 0 al recorrer una lista, por lo que la posición del primer elemento será la posición 0, la del segundo elemento de la lista será la posición 1 y, así sucesivamente.

Por ejemplo, si tenemos la lista `numeros = [4, 2, 4, 8, 6, 10]`, la sentencia `numeros.index[4]` nos devolverá el valor `0`, que es la primera vez que se encuentra el elemento que le hemos indicado, y la sentencia `números.index[10]` nos devolverá el valor `5`.

Si, por ejemplo, nuestra lista es `asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”, “Lengua”]`, la sentencia `asignaturas.index(“Ciencias Sociales”)` nos devolverá el valor `1`.

8. INSERTAR UN ELEMENTO EN UNA POSICIÓN CONCRETA

El método `insert()` nos permite insertar un elemento en una posición concreta en una lista. Funciona de la siguiente manera:

```
Nombre de la lista.insert(número del índice que corresponde con la posición en la que queremos insertar el elemento, elemento que queremos insertar)
```

La diferencia entre `append` e `insert` es que el primero siempre añade el elemento al final de la lista, mientras que el segundo te permite indicar la posición en la que lo quieres añadir.

Por ejemplo, si tenemos la lista `asignaturas = [“Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”]` y queremos añadir el elemento “Lengua”, pero al principio de la lista, tendríamos que utilizar la sentencia:

```
asignaturas.insert(0, “Lengua”)
```

Con esto le estamos diciendo que queremos insertar la palabra “Lengua” en la lista asignaturas en la primera posición, o en la posición cuyo índice vale 0. Con lo que nuestra lista ahora sería `asignaturas = [“Lengua”, “Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”]`

Con otros elementos funciona exactamente igual, si tenemos la lista `mezcla = [1, 7, “día”, 4, “noche”]` y queremos añadir el elemento 6 en la tercera posición, utilizaríamos la sentencia `mezcla.insert(2, 6)` con lo que obtendríamos `[1, 7, 6, “día”, 4, “noche”].`

9. INVERTIR UNA LISTA

Cuando queremos invertir la posición de todos los elementos de una lista utilizamos el método `reverse()`. Este método tampoco necesita que le indiquemos un parámetro, ya que actúa sobre todos los elementos. Funciona de la siguiente manera:

```
Nombre de la lista.reverse()
```

Veamos cómo funciona con un ejemplo. Si tenemos la lista mezcla anterior, `mezcla = [1, 7, 6, “día”, 4, “noche”]`, la sentencia `mezcla.reverse()` nos devolverá la lista `[“noche”, 4, “día”, 6, 7, 1]`.

10. ORDENAR UNA LISTA

El método `sort()` nos permite ordenar una lista de forma ascendente o descendente. Si nuestra lista contiene números los ordenará de menor a mayor, si queremos ordenarla de forma ascendente, o de mayor a menor, si queremos ordenarla de forma descendente. En cambio, si nuestra lista contiene palabras, las ordenará en alfabéticamente de forma ascendente o descendente según le indiquemos.

Funciona de la siguiente manera:

- Si queremos ordenar la lista de forma ascendente, la sentencia se construye así: `Nombre de la lista.sort(reverse=False)`
- Si queremos ordenar la lista de forma descendente, la sentencia se construye así: `Nombre de la lista.sort(reverse=True)`

Por ejemplo, si tenemos nuestra lista `numeros = [4, 2, 4, 8, 6, 10]`, la sentencia `numeros.sort(reverse=False)` nos devolverá la lista `[2, 4, 4, 6, 8, 10]`, y si tenemos la sentencia `números.sort(reverse=True)` nos devolverá la lista `[10, 8, 6, 4, 4, 2]`.

Si tenemos la lista `asignaturas = [“Lengua”, “Matemáticas”, “Ciencias Sociales”, “Ciencias Naturales”, “Geografía”]`, la sentencia `asignaturas.sort(reverse=False)` nos devolverá la lista `[“Ciencias Naturales”, “Ciencias Sociales”, “Geografía”, “Lengua”, “Matemáticas”]`, y si tenemos la sentencia `asignaturas.sort(reverse=True)` nos devolverá la lista `[“Matemáticas”, “Lengua”, “Geografía”, “Ciencias Sociales”, “Ciencias Naturales”]`.

Debéis tener en cuenta que las listas que tienen elementos de tipos diferentes, por ejemplo, la lista mezcla que teníamos anteriormente (`mezcla = [1, 7, 6, “día”, 4, “noche”]`), esta no se podrá ordenar, porque no hay forma de saber si un número es mayor que una palabra, o viceversa, por lo que este método solo se usa con listas que contienen el mismo tipo de elementos.

Si al método `sort` no le incluimos la condición reverse, es decir, si lo utilizamos de esta manera, nombre de la `lista.sort()`, ordenará la lista siempre de forma ascendente. Es decir, no incluirle el parámetro `reverse` es equivalente a incluir el parámetro `reverse=False`.

11. TUPLAS

Las tuplas son un tipo de datos que contienen conjuntos de elementos, igual que las listas, pero con la diferencia de que estas son inmutables, es decir, que una vez que se crean sus valores no se pueden modificar.

Las tuplas se crean introduciendo sus elementos entre paréntesis. Por ejemplo:

```
animal = (“perro”, “gato”, “pájaro”)

numero = (2, 4, 6, 8)
```

```
mezcla = (“gato”, 2, 4, “perro”)
```

Son tipos de datos que suelen utilizarse cuando nuestro conjunto de elementos está definido de ante mano y tiene un orden concreto, cuando sus elementos son conocidos. Por ejemplo, si hablamos de los meses del año, van de Enero a Diciembre y en ese orden, por lo tanto, para trabajar con ellos, podríamos guardarlos en una variable tipo tupla.

```
meses = ("Enero", "Febrero", "Marzo", "Abril", "Mayo", "Junio", "Julio", "Agosto", "Septiembre", "Octubre", "Noviembre", "Diciembre")
```

11.1. Métodos

En las tuplas no podemos utilizar todos los métodos que utilizábamos en las listas, ya que, como son inmutables, no podemos añadirles elementos, eliminar elementos, ordenarlos, etc.

Pero hay otros métodos como, por ejemplo, `count()`, que sí se puede utilizar para contar los elementos de una tupla. El siguiente código mostraría el número de veces que aparece el número 4 en la tupla `numero`.

```
numero = (2, 4, 6, 8, 4)

print(numero.count(4))
```

En este caso la respuesta sería `2`.

El método i`ndex()`, también se puede utilizar para conocer la posición, o el valor del índice, de un elemento de la tupla. Por ejemplo:

```
numero = (2, 4, 6, 8, 4)

print(numero.index(6))
```

Devolvería el número `2`, que es el valor de la posición en la que se encuentra el número 6.

12. CONJUNTOS O SETS

Los sets o conjuntos son colecciones de elementos no ordenados y únicos, es decir, en los conjuntos no hay elementos repetidos.

Al igual que en las listas y las tuplas, los elementos de un conjunto pueden ser del mismo tipo o no.

Los conjuntos se representan entre llaves. Por ejemplo:

```
animal = {“perro”, “gato”, “pájaro”}

numero = {2, 4, 6, 8}

mezcla = {“gato”, 2, 4, “perro”}
```

Para crear conjuntos, podemos hacerlo incluyendo directamente los elementos entre llaves, como veíamos en el ejemplo anterior, pero teniendo en cuenta que, aunque creemos conjuntos con elementos repetidos, solo se guardarán los elementos que no están repetidos y sin ningún orden en concreto.

Por ejemplo, nosotros podemos tener la siguiente declaración de una variable de tipo conjunto:

```
colores = {“rojo”, “verde”, “rojo”, “azul”, “verde”, “blanco”, “amarillo”, “azul”}
```

Pero, cuando le pidamos que nos muestre el conjunto colores por pantalla con la siguiente instrucción:

```
print(colores)
```

El resultado que obtendremos será:

```
{'verde', 'blanco', 'azul', 'rojo', 'amarillo'}
```

Como veis, no hay elementos repetidos y nos ha mostrado los elementos sin ningún tipo de orden. De hecho, si volvéis a ejecutarlo en un nuevo notebook, podría lanzar otro resultado diferente como `{'azul', 'blanco', 'amarillo', 'verde', 'rojo'}`.

12.1. Set

Hay otra forma de crear conjuntos que es utilizando el método `set()`. Este método se utiliza cuando queremos crear conjuntos desde, por ejemplo, una lista.

Por ejemplo, si tenemos la lista `colores = [“rojo”, “verde”, “rojo”, “azul”, “verde”, “blanco”, “amarillo”, “azul”]`, podemos crear un conjunto directamente así:

```
conjunto = set(colores)
```

O también así:

```
conjunto = set([“rojo”, “verde”, “rojo”, “azul”, “verde”, “blanco”, “amarillo”, “azul”])
```

Eso sí, al método `set` solo se le puede pasar un parámetro, en este caso una lista completa como la del segundo ejemplo se entiende como un parámetro.

También podemos hacer que una palabra, o una frase, se guarde como elementos separados de un conjunto utilizando `set`.

Por ejemplo, si tenemos la palabra “Hola”, podríamos tener la sentencia `conjunto = set(“Hola”)` lo que nos crearía el conjunto `{“o”, “H”, “a”, “l”}`. Recordad que no sigue ningún orden y que no guarda elementos repetidos.

Otro ejemplo, si tenemos `frase = set("estoy estudiando")`, que tiene letras repetidas, tendremos el conjunto `{'u', 'i', 'n', 't', 'o', 's', 'e', ' ', 'a', 'd', 'y'}`, sin repetidos y sin ordenar.

Básicamente, set nos permite crear un conjunto desde cualquier string.

12.2. Comprobar elementos

Cuando queremos comprobar si un elemento está en un conjunto, podemos comprobarlo directamente con el operador `in`.

Por ejemplo, si tenemos el conjunto `colores = {'verde', 'blanco', 'azul', 'rojo', 'amarillo'}` la condición `'verde' in colores`, será `True` si el elemento verde está en el conjunto colores y será `False` si no lo está. En este caso, y con este código:

```
colores = {'verde', 'blanco', 'azul', 'rojo', 'amarillo'}

print("verde" in colores)
```

Nos mostrará por pantalla `True`, porque el color verde está en el conjunto colores.

12.3. Añadir elementos

Para añadir elementos en un conjunto utilizamos el método `add()`. Este método se utiliza de la siguiente manera:

```
Nombre del conjunto.add(elemento a añadir)
```

Por ejemplo, si tenemos el conjunto `colores = {'verde', 'blanco', 'azul', 'rojo', 'amarillo'}` y queremos añadir el elemento "negro", lo haremos así:

```
colores.add('negro')
```

Con lo que nuestro conjunto será ahora `{'azul', 'blanco', 'amarillo', 'verde', 'rojo', 'negro'}`. Fijaos de nuevo en que el orden no tiene por qué ser el mismo que el que hemos definido al crearlo.

Si queremos añadir otro elemento que ya está en el conjunto como, por ejemplo, el color rojo, podremos hacerlo, pero nuestro conjunto no variará, porque el elemento ya está en él y no guarda los repetidos.

12.4. Eliminar elementos

Para eliminar elementos de un conjunto utilizamos el método `remove()`. Este método se utilizaría de forma similar a las listas, concretamente:

```
Nombre del conjunto.remove(elemento que queremos eliminar)
```

Por ejemplo, si tenemos el conjunto `colores = {'azul', 'blanco', 'amarillo', 'verde', 'rojo', 'negro'}` y queremos eliminar el color amarillo, tendríamos que hacerlo así:

```
colores.remove(“amarillo”)
```

Con lo que ahora nuestro conjunto sería `{'verde', 'blanco', 'azul', 'rojo', 'negro'}`

12.5. Combinar conjuntos

Los conjuntos también se pueden combinar entre ellos y hay dos métodos que nos permiten hacerlo, `union()` y `update()`. La diferencia entre ellos es que `union()` tiene que almacenar el conjunto resultado en un conjunto nuevo y `update()` nos permitirá sobrescribir alguno de los dos conjuntos que queremos combinar.

Veamos algunos ejemplos:

Si tenemos los conjuntos `colores = {'verde', 'blanco', 'azul', 'rojo', 'amarillo'}` y `numeros = {1, 2, 3, 6}` y utilizamos el método `union()`, tendremos que hacer los siguiente:

```
colores = {'verde', 'blanco', 'azul', 'rojo', 'amarillo'}

numeros = {1, 2, 3, 6}

mezcla = colores.union(numeros)
```

Ahora el conjunto `mezcla` será `{1, 2, 3, 'azul', 'blanco', 6, 'rojo', 'amarillo', 'verde'}`, y los conjuntos `colores` y `numeros` no variarán.

Sin embargo, si usamos el método `update()`, nuestro código será:

```
colores = {'verde', 'blanco', 'azul', 'rojo', 'amarillo'}

numeros = {1, 2, 3, 6}

colores.update(numeros)
```

Por lo tanto, `numeros` se mantendrá tal cual, no se modificará, pero `colores` ahora será `{1, 'rojo', 2, 3, 'azul', 'blanco', 6, 'amarillo', 'verde'}`

13. DICCIONARIOS

Los diccionarios son estructuras que almacenan conjuntos de pares (dos elementos) que contienen siempre una clave y un valor.

Son mutables y admiten, como las listas, las tuplas y los conjuntos, distintos tipos de datos.

Cada diccionario se representa entre llaves y la forma de crearlos sería:

```
Nombre del diccionario = {“clave1”: “valor1”, “clave2”: “valor2”, …, “clave_n”: “valor_n”}
```

Por ejemplo, si tenemos una agenda, cada persona podría estar guardada en un diccionario con estos datos:

```
persona = {“nombre”: “Ana”, “apellidos”: “Santo Santo”, “telefono”: 666666666}
```

Como veis, todas las claves son de tipo `string`, y los datos que contienen, en el primer y segundo elemento son de tipo `string` pero, en el último elemento, es de tipo `int`.

También podríamos tener un diccionario como este:

```
posicion = {1:’Ana’, 2:’Andrés’, 3:’Julia’, 4:’Agustín’}
```

Es decir, las claves también pueden ser de otro tipo diferente a `string`, en este caso son de tipo `int`.

Nos podemos encontrar, también, que una lista puede formar parte de un diccionario. Por ejemplo, si tenemos el diccionario `joyas = {“tipo”:”anillo”, “modelo”:”reina”, “materiales”:[“oro blanco”, “oro rosa”]}`, la clave `“materiales”` tiene como valor la lista `[“oro blanco”, “oro rosa”]`.

Un diccionario, a su vez, también puede contener otro diccionario. Por ejemplo, si tenemos el diccionario `agenda = {“identificador”: 1, “datos”:{“nombre”: “Ana”, “apellidos”: “Santo Santo”, “telefono”: 666666666}}`, la clave `“datos”` es un diccionario con diferentes pares clave, valor.

En este último caso, como el elemento de la clave `“datos”` coincide con nuestro diccionario `persona`, también podríamos definir el diccionario agenda así:

```
agenda = {“identificador”: 1, “datos”:persona}
```

Y esto nos devolvería el mismo resultado, es decir, que `agenda` es `{“identificador”: 1, “datos”:{“nombre”: “Ana”, “apellidos”: “Santo Santo”, “telefono”: 666666666}}`

13.1. Acceder a los datos

Podemos acceder a cualquiera de los elementos de un diccionario a través de su clave. Funciona de la siguiente manera:

```
Nombre del diccionario[clave] devuelve valor
```

Por ejemplo, en los diccionarios anteriores, tendríamos que:

`persona[“apellidos”]` nos devolverá `“Santo Santo”`

`persona[“telefono”]` nos devolverá `666666666`

`posicion[1]` nos devolverá `“Ana”`

`posicion[3]` nos devolverá `“Julia”`

`joyas[“materiales”]` nos devolverá `[“oro blanco”, “oro rosa”]`

`agenda[“datos”]` nos devolverá `{“nombre”: “Ana”, “apellidos”: “Santo Santo”, “telefono”: 666666666}`

13.2. Añadir elementos

Para añadir elementos a un diccionario, solo tenemos que identificar la clave y darle un valor. Se haría de la siguiente manera:

```
Nombre del diccionario[clave a añadir] = valor de dicha clave
```

Por ejemplo, si en el diccionario `persona` queremos añadir un nuevo par, cuya clave sea `email`, podríamos hacerlo así:

```
persona[“email”] = “ana@python.com”
```

Nuestro diccionario persona será ahora `{'nombre': 'Ana', 'apellidos': 'Santo Santo', 'telefono': 666666666, 'email': 'ana@python.com'}`

Si queremos añadir en nuestro diccionario `posicion` un cuarto elemento, podríamos hacerlo así:

```
posicion[4] = “Sofía”
```

Con lo que nuestro diccionario `posicion` ahora será `{1: 'Ana', 2: 'Andrés', 3: 'Julia', 4: 'Sofía'}`

13.3. Sustituir elementos

Para actualizar o sustituir cualquier elemento, también lo hacemos identificando la clave y dándole un nuevo valor al elemento. Su estructura es la siguiente:

```
Nombre del diccionario[nombre de la clave cuyo elemento queremos actualizar] = nuevo valor del elemento
```

Por ejemplo, si en nuestro diccionario `persona` queremos actualizar el número de teléfono de Ana, lo haríamos así:

```
persona[“telefono”] = 222222222
```

Con lo que nuestro diccionario ahora sería `{'nombre': 'Ana', 'apellidos': 'Santo Santo', 'telefono': 222222222, 'email': 'ana@python.com'}`

Si en nuestro diccionario `joyas` queremos modificar los elementos con clave `“materiales”` haríamos lo siguiente:

```
joyas[“materiales”] = [“oro rosa”, “plata”]
```

Con lo que nuestro diccionario sería ahora `{'tipo': 'anillo', 'modelo': 'reina', 'materiales': ['oro rosa', 'plata']}`

13.4. Eliminar elementos

Para eliminar elementos de un diccionario podemos utilizar la función `pop()` o la palabra reservada `del`. Veamos cómo se utiliza cada una de ellas.

La estructura para utilizar la función `pop()` es la siguiente:

```
Nombre del diccionario.pop(nombre de la clave del elemento que queremos eliminar)
```

Por ejemplo, si de nuestro diccionario `joyas` queremos eliminar el elemento con clave `“modelo”`, lo haríamos de la siguiente forma:

```
joyas.pop(“modelo”)
```

Con lo que nuestro diccionario sería ahora `{'tipo': 'anillo', 'materiales': ['oro rosa', 'plata']}`

La palabra reservada `del` tiene una estructura diferente,

```
del nombre del diccionario[nombre de la clave del elemento que queremos eliminar]
```

Si tenemos el mismo caso anterior, `joyas = {'tipo': 'anillo', 'modelo': 'reina', 'materiales': ['oro rosa', 'plata']}`, eliminaríamos el elemento clave “modelo” de la siguiente forma:

```
del joyas[“modelo”]
```

Con lo que obtendríamos el mismo resultado, `{'tipo': 'anillo', 'materiales': ['oro rosa', 'plata']}`

13.5. Recorrer diccionarios

Para recorrer un diccionario podemos utilizar un bucle `for`. Lo haríamos de la siguiente manera:

```
for índice in nombre del diccionario:
      acción a ejecutar
```

Aquí índice irá tomando en cada iteración el valor de la clave.

Por ejemplo, en nuestro diccionario `joyas = {'tipo': 'anillo', 'modelo': 'reina', 'materiales': ['oro rosa', 'plata']}`, el fragmento de código siguiente nos permitirá ver qué valor tiene `i` en cada iteración:

```
joyas = {'tipo': 'anillo', 'modelo': 'reina', 'materiales': ['oro rosa', 'plata']}

for i in joyas:

      print(i)
```

Nos devolverá:

```
tipo
modelo
materiales
```

Esto quiere decir que, en la primera iteración `i` tomará el valor `tipo`, en la segunda el valor `modelo` y en la tercera el `valor materiales`.

Aunque también podemos acceder al valor que acompaña a cada clave. Por ejemplo, si tenemos el siguiente código:

```
joyas = {'tipo': 'anillo', 'modelo': 'reina', 'materiales': ['oro rosa', 'plata']}

for i in joyas:

      print(joyas[i])
```

Nos devolverá:

```
anillo
reina
['oro blanco', 'oro rosa']
```

14. MÁS INFORMACIÓN

En este vídeo puedes ver cómo se trabaja con listas. Como verás, el ejemplo se basa en una función, pero nosotros aún no hemos estudiado cómo crear funciones, aunque no te preocupes por esa parte, intenta entender cómo funciona cada uno de los apartados en los que se crea una lista, se muestra, se añaden elementos, se eliminan, etc.:

Vídeo: https://www.youtube.com/watch?v=MHvrJhshEU0

En este vídeo puedes ver cómo se trabaja con listas y tuplas:

Vídeo: https://www.youtube.com/watch?v=0NTaCJQUE1I

También, en estos links, podrás ampliar la información sobre los métodos que puedes utilizar a la hora de trabajar con conjuntos, ya que nosotros hemos estudiado los más básicos que necesitarás para poder trabajar con ellos.

Amplia información: https://ellibrodepython.com/sets-python

En el siguiente enlace puedes ampliar la información sobre los métodos que puedes utilizar con los diccionarios:

Ampliar información: https://ellibrodepython.com/diccionarios-en-python

Y en el siguiente vídeo puedes ver algunos ejemplos de ejercicios resueltos con diccionarios:

Vídeo: https://www.youtube.com/watch?v=uOpW1tKKO8M