El futuro digital es de todos

MinTIC

Universidad Industrial de Santander

MisiónTIC 2022

ESTRUCTURAS DE DATOS

El futuro digital es de todos

MinTIC

ESTRUCTURA DE DATOS EN PYTHON

Universidad Industrial de Santander

MisiónTIC2022

INTRODUCCIÓN

Las estructuras de datos son agrupaciones de variables y funciones simples que conforman el conjunto de datos más complejo. Para ayudar a gestionar problemas complejos y tipos de datos complejos, los informáticos suelen trabajar con abstracciones. Una **abstracción** es un mecanismo para separar las propiedades de un objeto y restringir el enfoque a aquellas relevantes en el contexto actual. El usuario de la abstracción no tiene que comprender todos los detalles para poder utilizar el objeto, sino sólo los relevantes para la tarea o problema actual. Un ejemplo cotidiano de abstracción es un carro, del cual no necesitamos saber los detalles internos del funcionamiento del motor para manejarlo o transportarnos.

Dos tipos comunes de abstracciones que se encuentran en la informática son la abstracción **procedimental o funcional y la abstracción de datos.**

1.1. La abstracción procedimental

Es el uso de una función o método sabiendo lo que hace, pero ignorando cómo se logra. Considere la función matemática de raíz cuadrada que probablemente haya utilizado en algún momento. Sabes que la función calculará la raíz cuadrada de un número dado, pero ¿sabes cómo se calcula la raíz cuadrada? ¿Importa si sabe cómo se calcula o simplemente saber cómo utilizar correctamente la función es suficiente?

1.2. La abstracción de datos

Es la separación de las propiedades de un tipo de datos (sus valores y operaciones) de la implementación de ese tipo de datos. Has usado cadenas de caracteres o Strings en Python muchas veces. Pero ¿sabes cómo se implementan? Es decir, ¿sabes cómo se estructuran los datos internamente o cómo se implementan las distintas operaciones?

Material de estudio complementario

Links:

https://www.youtube.com/watch?v=TOa40RJYAuM

https://www.youtube.com/watch?v=PNR_Uq8RkP4

Referencias Bibliográficas

- Necaise, R. D. (2010). *Data Structures and Algorithms Using Python*. Wiley Publishing.

El futuro digital es de todos

MinTIC

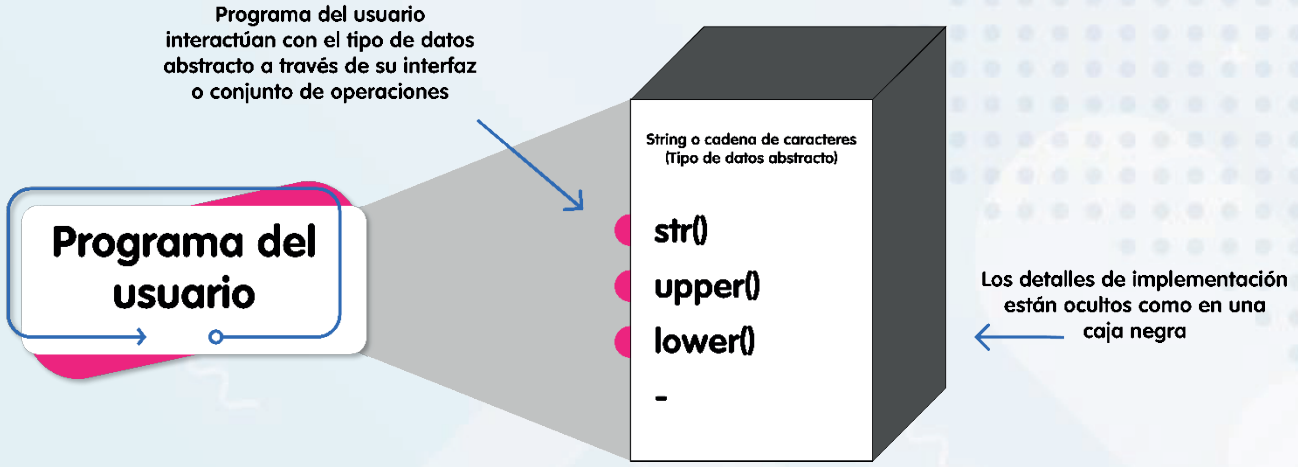
TIPOS DE DATOS ESTRUCTURADOS

Universidad Industrial de Santander

MisiónTIC2022

Un tipo de datos abstracto es un tipo de datos definido por Python que especifica un conjunto de valores de datos y una colección de operaciones bien definidas que se pueden realizar en esos valores. Los tipos de datos abstractos se definen independientemente de su implementación, lo que nos permite centrarnos en el uso del nuevo tipo de datos en lugar de en cómo se implementa. Esta separación generalmente se aplica al requerir la interacción con el tipo de datos abstracto a través de una **interfaz** o un conjunto definido de operaciones.

Los tipos de datos abstractos pueden verse como cajas negras como se ilustra en la figura. Los programas de usuario interactúan con instancias de un tipo de datos abstracto invocando una de las varias operaciones definidas por su interfaz.



La implementación de las diversas operaciones está oculta dentro de la caja negra, cuyo contenido no tenemos que conocer para utilizar el tipo de datos abstracto. Existen varias ventajas de trabajar con tipos de datos abstractos y centrarse en el "qué", en lugar del "cómo".

- **Podemos concentrarnos en resolver el problema en cuestión en lugar de atascarnos en los detalles de implementación**. Por ejemplo, suponga que necesitamos extraer una colección de valores de un archivo en el disco y almacenarlos para su uso posterior en nuestro programa. Si nos centramos en los detalles de implementación, entonces tenemos que preocuparnos por qué tipo de estructura de almacenamiento usar, cómo debe usarse y si es la opción más eficiente.

- Podemos reducir los errores lógicos que pueden ocurrir por el uso indebido de estructuras de almacenamiento y tipos de datos al evitar el acceso directo a la implementación.

- Es más fácil administrar y dividir programas más grandes en módulos más pequeños, lo que permite que diferentes miembros de un equipo trabajen en módulos separados.

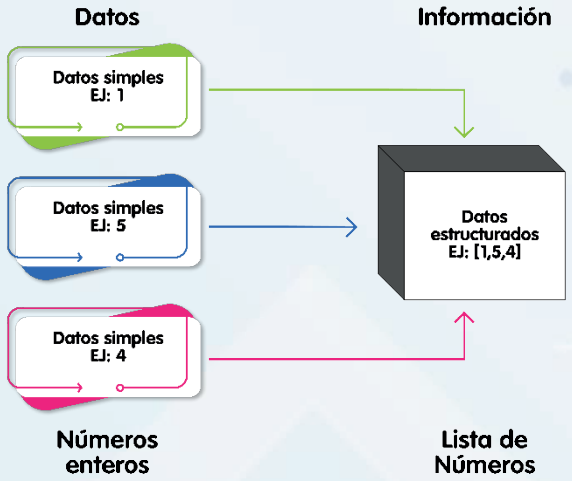
Trabajar con tipos de datos abstractos, que separan la definición de la implementación, es ventajoso para resolver problemas y escribir programas. Los tipos de datos abstractos proporcionados en lenguajes de programación, como Python, son implementados por los encargados del lenguaje. Los tipos de datos abstractos pueden ser simples o complejos.

2.1. Tipo de datos abstracto simple

Se compone de uno o varios campos de datos nombrados individualmente, como los que se utilizan para representar una fecha o un número racional.

2.2. Tipos de datos abstractos complejos

Se componen de una colección de valores de datos como listas o diccionarios en Python. Los tipos de datos abstractos complejos se implementan utilizando una estructura de datos particular, que es la representación física de cómo se organizan y manipulan los datos. Las estructuras de datos se pueden caracterizar por cómo almacenan y organizan los elementos de datos individuales y qué operaciones están disponibles para acceder y manipular los datos. Como puede ser un conjunto de números, una cadena de caracteres, letras, listas o secuencias.



Material de estudio complementario

https://www.youtube.com/watch?v=sRvUxw-F54c

https://www.youtube.com/watch?v=Kcnp3e17Gq4

Referencias Bibliográficas

- Necaise, R. D. (2010). *Data Structures and Algorithms Using Python*. Wiley Publishing.

El futuro digital es de todos

MinTIC

LISTAS Y TUPLAS

Universidad Industrial de Santander

Las **tuplas y listas** como estructura de datos en Python se refieren a un **área de almacenamiento adyacente o contiguo en memoria** donde podemos reunir o guardar distintos tipos de elementos en un mismo lugar o variable. Las listas son estructuras que permiten modificarse a través de métodos siendo mutable. Mientras que las tuplas son inmutables.

Es importante recordar que las variables **mutables** son variables que se pueden modificar mientras nuestro código o script está funcionando, es decir, en tiempo de ejecución. Mientras que las **inmutables**, sólo podemos modificarlas antes de que nuestro programa o script esté funcionando o en ejecución.

3.1. Listas

Para crear una **lista** en Python lo hacemos declarando una variable y añadiendo los elementos o ítems usando corchetes cuadrados:

```
Objetos = ["Celular", "Vehículo", "Escritorio"]

Numeros = [1, 22, 333, 4444, 55555]
```

Como vimos en el ejemplo anterior, creamos dos listas "`Objetos`" y "`Numeros`", la primera lista contiene ítems de tipo texto o cadena de caracteres (String). Es por esto que **deben ser separados por coma e ir entre comillas cada uno.**

Es posible crear listas vacías sin especificar sus elementos, ya que al ser mutables sus elementos o ítems pueden ser agregados posteriormente.

Para crear una lista sin elementos, la debemos declarar de la misma forma pero sin elementos, siempre teniendo en cuenta la sintaxis de los **corchetes cuadrados.**

```
Lista_vacia = []
```

3.2. Tuplas

Una **tupla** es similar a una lista, pero con diferencias que es importante tener en cuenta.

Las tuplas no se pueden **modificar** durante la ejecución del código ya que son **inmutables**. Es por esto que en **el momento de su creación se deben especificar todos los elementos que se van a guardar**. A diferencia de las listas, las tuplas usan "paréntesis" para definir sus elementos. A continuación se muestran algunos ejemplos de tuplas.

```
Elementos = ("Apartamento", "Ascensor", "Batería", "Libro", "Alfombra")

Numeros = (1, 2, 3, 4, 55)
```

Se han creado dos tuplas de forma independiente, "`Elementos`" y "`Numeros`", la primera tupla contiene ítems de texto o cadena de caracteres (string). Por esta razón, **deben estar divididos por coma y entre comillas**

Es importante recordar que cuando se crean las tuplas se debe, de forma obligatoria especificar cuáles elementos se van a guardar ya que no vamos a poder modificarlos en tiempo de ejecución.

3.3. Accediendo a los elementos de listas y tuplas

Para acceder a los elementos se debe tener en cuenta el índice o posición al que pertenece el elemento a acceder. Un index o en español índice es un número entero que ubica el ítem dentro del grupo de ítems de una lista o tupla. El primer índice es 0 para la primera posición.

La lista creada:

```
Objetos = ["Celular", "Vehículo", "Escritorio"]
```

se puede representar con sus posiciones así:

Objetos =	[“Celular”,	“Vehículo”,	“Escritorio”]
	0	1	2

```
Numeros = [1, 22, 333, 4444, 55555]
```

Numeros =	[1,	22,	333,	444,	55555]
	0	1	2	3	4

Entonces si se desea acceder a la posición "1" de la Lista "`Numeros`" **se debe especificar la lista y seguido entre corchetes cuadrados el número de la posición que se requiere acceder:**

```
Numeros[0]
```

Resultado: 1

Ya que se eligió el índice 0, el resultado debería mostrar el número 1. En la otra lista, la llamada "`Objetos`", se accedería de la misma forma para obtener el string "`Celular`".

```
Objetos[0]
```

Resultado: Celular

Si se requiriera hacer el mismo proceso para la estructura de tuplas, se realizaría de la misma manera **usando corchetes cuadrados**

```
Elementos [2]
```

Resultado: Batería

Ya que se decidió obtener el ítem del índice 2, nos debe mostrar el string "Bateria".

```
Numeros [2]
```

Resultado: 3

Y aquí se imprime el número 3 que corresponde a la posición o índice 2. El posicionamiento en python empieza con el "0".

Como se pudo observar en el momento de acceder a cualquiera de las dos estructuras, **ya sean listas o tuplas, se deben usar corchetes cuadrados**. Sólo usamos paréntesis cuando se crea la tupla. Es importante aclarar que si se intenta acceder a un índice mayor al último índice del último elemento ocurrirá un error.

3.4. Acceder a más de un elemento en tuplas y listas

Para acceder a más de un elemento se tienen que especificar una posición o índice de inicio y uno final. Por ejemplo, para la lista "`Numeros`":

```
Numeros [0:3]
```

```
Resultado: 1, 22, 333
```

En esta ocasión, se definió un rango separado por dos puntos (:) . El número antes de los dos puntos, es decir el 0, define el comienzo del rango y el número después de los dos puntos, es decir, el 3, define el final del rango. Si cambiamos el rango, se cambian los elementos que se obtienen.

```
Numeros [1:3]
```

```
Resultado: 22, 333
```

Ya sea una lista o tupla, en Python podemos mostrar todos los elementos excepto algunos elementos del final. Por ejemplo:

```
Numeros [:-2]
```

```
Resultado: 1, 22, 333
```

En esta ocasión no se colocó un número de inicio que iría antes de los " : " y como índice final se colocó un número negativo, entonces, mostrará todos los elementos de la lista a excepción de los últimos 2.

3.5. Modificar listas en tiempo de ejecución

Podemos crear una lista y luego modificar sus elementos mientras se ejecuta el código. Para esto las listas tienen un conjunto de funciones o métodos que realizan acciones y operaciones sobre una lista en particular.

```
Nombres = ['David', 'Elkin', 'Juliana', 'Silvia']
print(Nombres) # Resultado ['David', 'Elkin', 'Juliana', 'Silvia']
Nombres.append ('Ivon')
print(Nombres) # Resultado ['David', 'Elkin', 'Juliana', 'Silvia', 'Ivon']
```

append permite añadir un ítem al final de una lista en Python. Como lo muestra el código anterior, solo es necesario seleccionar la lista a editar seguido de un punto con la palabra clave append y finalmente el elemento que se requiere añadir entre paréntesis como argumento de la función o método.

Es necesario crear una lista antes de modificarla.

El método extend se utiliza para agregar elementos interables como un string u otra lista con sus elementos.

```
Nombres = ['David', 'Elkin', 'Juliana', 'Silvia']

print(Nombres) # Resultado ['David', 'Elkin', 'Juliana', 'Silvia']

Nombres.extend('MARIA')

print(Nombres)      #      Resultado      ['David',      'Elkin',      'Juliana',
'Silvia','M','A','R','I','A']
```

Un ejemplo de agregar una lista a otra lista:

```
Numeros1 = [1, 2, 3]

Numeros2 = [4, 5, 6]

Numeros1.extend(Numeros2) #Añadimos la lista Numeros2 como extensión de la lista
Numeros1

print(Numeros1) # Resultado [1, 2, 3, 4, 5, 6]
```

El método insert permite añadir un elemento en una posición o índice específico.

```
Nombres = ['David', 'Elkin', 'Juliana', 'Silvia’]

print(Nombres) # Resultado ['David', 'Elkin', 'Juliana', 'Silvia']

Nombres.insert(1, 'Ivon')

print(Nombres) #Resultado ['David', 'Ivon', 'Elkin', 'Juliana', 'Silvia']
```

Como se pudo observar, el elemento ‘Ivon’ se añadió al índice 1 tal como lo indicamos y actualizó los índices de los demás elementos. También podemos remover ítems de una lista de Python

```
Nombres = ['David', 'Elkin', 'Juliana', 'Silvia']

print(Nombres) #Resultado ['David', 'Elkin', 'Juliana', 'Silvia']

Nombres.pop(1)

print(Nombres) #Resultado ['David', 'Juliana', 'Silvia']
```

En este caso el elemento con índice 1 (‘Elkin’) se elimina de la lista. Si a la función o método pop no se le indica el la posición, se eliminará el último elemento de la lista.

Si fuera requerido remover un ítem de una lista basado en el valor, el método remove se podría utilizar de la siguiente manera.

```
Nombres = ['David', 'Elkin', 'Juliana', 'Silvia']

print(Nombres) #Resultado ['David', 'Elkin', 'Juliana', 'Silvia']

Nombres.remove('Elkin')

print(Nombres) #Resultado ['David', 'Juliana', 'Silvia']
```

De esta manera, el elemento con el valor “Elkin” se elimina de la lista. Si el valor a eliminar no se encuentra en la lista se mostrará un error.

Existen maneras de crear listas de manera más fácil y efectiva. Range() crea un conjunto de valores según los parámetros de inicio, final y paso.

```
x = list(range(2, 20, 2))

print(x) # Resultado [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

En este ejemplo usamos list() para transformar el rango de valores en una lista. Respecto a range(I,F,S), el primer parámetro I=2 inicia la secuencia en 2. El segundo parámetro F=20 termina la secuencia hasta el número 20 sin incluirlo, es decir hasta 19. Finalmente, el tercer parámetro S=2 será el paso. La secuencia final irá de 2 hasta 20 de 2 en 2.

Si se necesitara crear una secuencia similar, pero de 20 hasta 2 de 2 en 2 sería de la siguiente manera:

```
x = list(range(20, 0, -2))

print(x) #Resultado [20, 18, 16, 14, 12, 10, 8, 6, 4, 2]
```

Material de estudio complementario

Links:

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

https://www.youtube.com/watch?v=xdCJa2QXmJ8

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

Referencias Bibliográficas

- Pythones, "Listas, Tuplas y Range en Python 3: Similitudes y diferencias" [Online]. Recuperado de: *https://pythones.net/listas-tuplas-python/*.

El futuro digital es de todos

MinTIC

CONJUNTOS

Universidad Industrial de Santander

MisiónTIC2022

Un conjunto es una colección desordenada de elementos. Cada elemento del conjunto es único (sin duplicados) y debe ser inmutable (no se puede cambiar).

Sin embargo, un conjunto en sí mismo es mutable. Podemos agregarle o quitarle elementos completos.

Los conjuntos también se pueden utilizar para realizar operaciones de conjuntos matemáticos como unión, intersección, diferencia simétrica, etc.

Un conjunto se crea colocando todos los elementos (elementos) entre llaves {}, separados por comas, o usando la función incorporada set(). Puede tener cualquier número de elementos y pueden ser de diferentes tipos (entero, flotante, tupla, cadena, etc.). Sin embargo, un conjunto no puede tener elementos mutables como listas, conjuntos o diccionarios como elementos.

```
mi_conjunto = {1, 2, 3}

print(mi_conjunto)# Resultado: {1, 2, 3}
```

```
mi_conjunto = {1.0, "Hello", (1, 2, 3)}

print(mi_conjunto) #Resultado: {1.0, (1, 2, 3), 'Hello'}
```

```
mi_conjunto = {1, 2, 3, 4, 3, 2}

print(mi_conjunto) # Resultado (No pueden tener duplicados): {1, 2, 3, 4}
```

```
mi_conjunto = set([1, 2, 3, 2])

print(mi_conjunto) # Resultado: {1, 2, 3}
```

```
mi_conjunto = {1, 2, [3, 4]} # Mostrará un error
```

La última línea mostrará un error ya que no podemos tener elementos mutables en un conjunto como lo es una lista. Es importante recordar que un conjunto no tiene un orden interno, por lo tanto, las posiciones de los elementos no importan y pueden variar entre la definición y la impresión del conjunto.

La forma de inicializar un conjunto vacío es con la función auxiliar set(). No debe usarse {} ya que las llaves vacías definen otro tipo de estructura. Los conjuntos son mutables, sin embargo, dado que están desordenados, la indexación no tiene ningún significado. No podemos acceder o cambiar un elemento de un conjunto mediante índices. Podemos agregar un solo elemento usando el método add() y varios elementos usando el método update(). El método update() puede tomar tuplas, listas, cadenas u otros conjuntos como argumento para añadir varios elementos individuales. En todos los casos se evitan los duplicados.

```
mi_conjunto = {1, 3}

print(mi_conjunto) # Resultado: {1, 3}
```

```
mi_conjunto.add(2)

print(mi_conjunto) # Resultado: {1, 2, 3}



mi_conjunto.update([2, 3, 4])

print(mi_conjunto) # Resultado: {1, 2, 3, 4}



mi_conjunto.update([4, 5], {1, 6, 8})

print(mi_conjunto) # Resultado: {1, 2, 3, 4, 5, 6, 8}
```

Un elemento en particular se puede eliminar de un conjunto usando los métodos discard() y remove(). La única diferencia entre los dos es que la función discard() deja un conjunto sin cambios si el elemento no está presente en el conjunto. Por otro lado, la función remove() generará un error en tal condición (si el elemento no está presente en el conjunto).

```
mi_conjunto = {21, 17, 0, 1, 60}

print(mi_conjunto )# Resultado: {21, 17, 0, 1, 60}
```

```
mi_conjunto.discard(1)

print(mi_conjunto) # Resultado: {21, 17, 0, 60}
```

```
mi_conjunto.remove(60)

print(mi_conjunto) # Resultado: {21, 17, 0}


mi_conjunto.discard(15)

print(mi_conjunto) # Resultado: {21, 17, 0}


mi_conjunto.remove(15)     #     Error.     El     elemento     15     no     existe.
```

De manera similar, podemos eliminar y devolver el primer elemento usando el método pop(). Dado que el conjunto es un tipo de datos desordenado, no hay forma de determinar qué elemento aparecerá. Es completamente arbitrario.

También podemos eliminar todos los elementos de un conjunto usando el método clear().

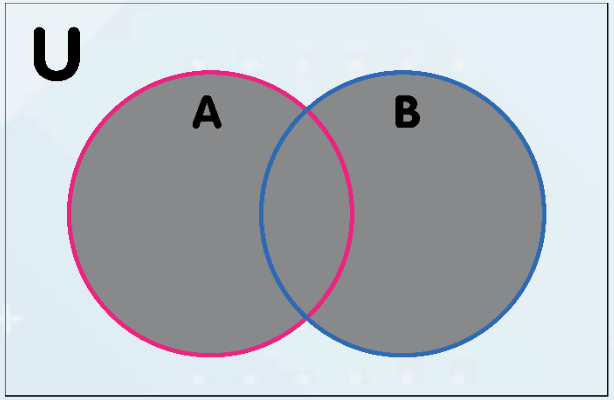
```
mi_conjunto = set("HolaMundo")

print(mi_conjunto) # Resultado: {'l', 'o', 'n', 'M', 'd', 'u', 'H', 'a'}



print(mi_conjunto.pop()) # Resultado: l
```

```
mi_conjunto.pop()

print(mi_conjunto) # Resultado: {'n', 'M', 'd', 'u', 'H', 'a'}



mi_conjunto.clear()

print(mi_conjunto) # Resultado: set()
```

4.1. Operaciones en conjunto

Los conjuntos se pueden utilizar para realizar operaciones de conjuntos matemáticos como unión, intersección, diferencia y diferencia simétrica. Podemos hacer esto con operadores o métodos.

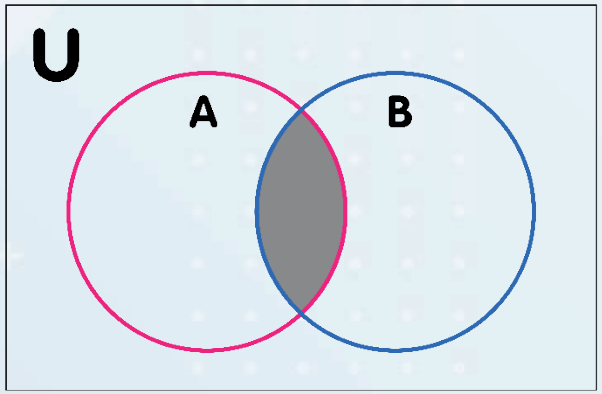
La unión de A y B es un conjunto de todos los elementos de ambos conjuntos.



La unión se realiza utilizando | operador. Lo mismo se puede lograr usando el método union().

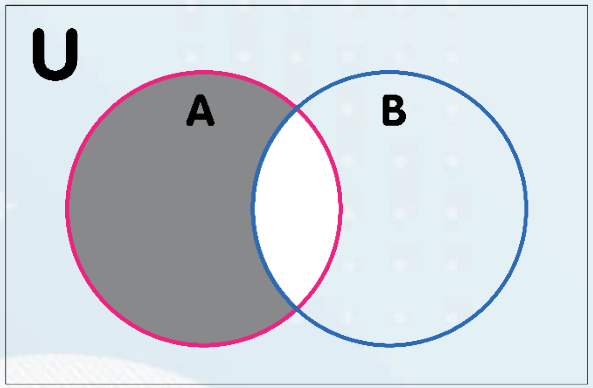
```
A = {1, 2, 3, 4, 5}
B = {4, 5, 6, 7, 8}
print(A | B) # Resultado: {1, 2, 3, 4, 5, 6, 7, 8}
print(A.union(B)) # Resultado: {1, 2, 3, 4, 5, 6, 7, 8}
print(B.union(A))
```

La intersección de A y B es un conjunto de elementos que son comunes en ambos conjuntos. La intersección se realiza mediante el operador. Lo mismo se puede lograr usando el método intersection().



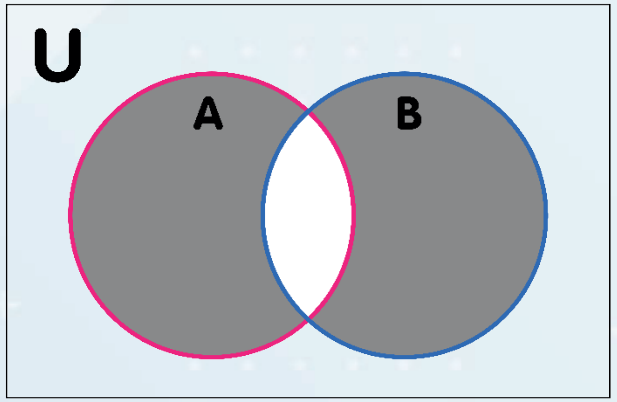
```
A = {0, 8, 3, 4, 5}
B = {1, 2, 6, 0, 8}

print(A & B) # Resultado: {0, 8}
print(A.intersection(B)) # Resultado: {0,8}
print(B.intersection(A)) # Resultado: {0,8}
```

La diferencia del conjunto B del conjunto A, (A - B) es un conjunto de elementos que están solo en A pero no en B. De manera similar, B - A es un conjunto de elementos que están en B pero no en A. La diferencia se realiza mediante el operador - . Lo mismo se puede lograr usando el método difference().



```
A = {0, 8, 3, 4, 5}
B = {1, 2, 6, 0, 8}


print(A - B) # Resultado: {3, 4, 5}
print( A.difference(B)) # Resultado: {3, 4, 5}
print( B.difference(A)) # Resultado: {1, 2, 6}
```

La diferencia simétrica de A y B es un conjunto de elementos que están en A y B pero no en ambos (excluyendo la intersección). La diferencia simétrica se realiza mediante el operador ^. Lo mismo se puede lograr usando el método symmetric_difference().



A = {0, 8, 3, 4, 5}

B = {1, 2, 6, 0, 8}

```
print(A ^ B) # Resultado: {1, 2, 3, 4, 5, 6}

print(A.symmetric_difference(B)) # Resultado: {1, 2, 3, 4, 5, 6}

print(B.symmetric_difference(A)) # Resultado: {1, 2, 3, 4, 5, 6}
```

Material de estudio complementario

Links:

https://www.youtube.com/watch?v=rmRrvol4XcM

https://www.youtube.com/watch?v=GdUwYO4Ru_s

https://www.youtube.com/watch?v=pPDQ4yzSuiU

Referencias Bibliográficas

- Necaise, R. D. (2010). *Data Structures and Algorithms Using Python*. Wiley Publishing.
- j2logo, "set python – Conjuntos en Python: El tipo set y operaciones más comunes" [Online]. Recuperado de: *https://j2logo.com/python/tutorial/tipo-set-python/*.

El futuro digital es de todos

MinTIC

DICCIONARIOS

Universidad Industrial de Santander

MisiónTIC2022

Los diccionarios nos permiten almacenar **valores** ordenados mediante **claves**. Esto nos permite ordenar datos de manera más natural y sencilla.

Para crear un diccionario entre llaves { } agrupamos **Clave** : **Valor** separadas por comas, así:

```
mi_diccionario = {'nombre': 'david', 'edad': 27, 'genero': 'masculino'}
```

En el código anterior las claves tienen **negritas** y sin negritas los valores. Utilizamos comillas simples para aquellos datos que son texto.

```
mi_informacion = {

'nombre' : 'David',

'apellido' : 'Cortez',

'sobrenombre' : 'DD',
```

```
'padres' : ["Marín gomez", "Julián Cortez"],

'edad' : 27,

'genero' : 'masculino',

'estado Civil' : 'Soltero',

'hijos' : 2,

'mascotas' : 'Perro',

'nombres de mascotas' : ["chato"]

}
```

Es posible almacenar en un valor de una clave elementos complejos como listas. Una vez que almacenamos los datos en el diccionario vamos a acceder a ellos.

```
print(mi_informacion['Apellido']) # Resultado: Cortez

print(mi_informacion.get('Apellido', "No tiene un apellido")) #Resultado: Cortez

print(mi_informacion.get('celular', "No tiene un celular")) #Resultado: No tiene
un celular
```

Con el método get() de un diccionario podemos obtener el valor de una clave pero si no existe la clave devolver un mensaje en string como respuesta.

Para agregar valores nuevos a un diccionario se debe realizar de la siguiente forma:

```
mi_informacion['salario'] = 200000
```

De esta forma, la clave será “salario” y el valor 200000. Si ahora queremos modificar un valor ya existente se debe realizar de la siguiente forma:

```
mi_informacion['edad'] = 38
```

Al ser una clave existente se modificará sólo el valor. Si se requiere eliminar una clave con su valor se debe utilizar:

```
mi_informacion.pop('hijos')

del(mi_informacion['mascotas'])
```

En la primera línea utilizamos pop() como una operación del diccionario y en la segunda línea usamos la función predefinida del(). De esta manera se habrá eliminado las **claves** de hijos y mascotas con sus respectivos **valores**.

Para listar todas las claves que tiene un diccionario podemos utilizar el método keys() de un diccionario en particular.

```
mi_diccionario = {'nombre': 'david', 'edad': 27, 'genero': 'masculino'}

print(mi_diccionario.keys()) #Resultado: dict_keys(['nombre', 'edad', 'genero'])
```

Finalmente, para listar todos los valores que tiene un diccionario podemos utilizar el método values() de un diccionario en particular.

Material de estudio complementario

Links:

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

https://www.youtube.com/watch?v=ZRxj3euWzuI

https://www.youtube.com/watch?v=_UELgsIxE7g

Referencias Bibliográficas

- Pythones, "Listas, Tuplas y Range en Python 3: Similitudes y diferencias" [Online]. Recuerpado de: *https://pythones.net/listas-tuplas-python/*.