1. TIPOS DE DATOS COMPLEJOS

Un tipo de dato complejo es un tipo de dato que se utiliza para representar estructuras de datos más complejas que los tipos de datos básicos como enteros, flotantes, cadenas de texto, etc.

1. 1.1. LISTAS

En Python, una lista es un tipo de dato que se utiliza para almacenar una colección ordenada y mutable de elementos (se refiere a la capacidad de un objeto de ser modificado después de haber sido creado). Puedes pensar en una lista como una secuencia de elementos que se pueden acceder, modificar y recorrer.

Para crear una lista en Python, se utilizan corchetes [] y los elementos se separan por comas. Ejemplo de cómo crear una lista con algunos elementos:

```
frutas = ["manzana", "banana", "naranja", "pera"]
```

Podés acceder a los elementos de una lista utilizando su índice. Los índices comienzan desde cero para el primer elemento y van aumentando en uno para cada elemento subsiguiente. Por ejemplo, para acceder al segundo elemento de la lista frutas, se utiliza frutas[1]:

```
print(frutas[1])  # Imprime "banana"
```

También podés modificar los elementos de una lista asignando un nuevo valor a un índice específico:

```
frutas[2] = "mandarina"

print(frutas)  # Imprime ["manzana", "banana", "mandarina",
"pera"]
```

Además, las listas en Python tienen varios métodos incorporados que se pueden utilizar para realizar operaciones comunes, como agregar elementos, eliminar elementos, ordenar la lista, etc. Aquí tienes algunos ejemplos de uso de estos métodos:

```
# Agregar elementos a una lista frutas.append("uva")  # Agrega

"uva" al final de la lista

print(frutas)  # Imprime ["manzana", "banana", "mandarina",
"pera", "uva"]


# Eliminar elementos de una lista frutas.remove("banana")  #

Elimina el elemento "banana"

print(frutas)  # Imprime ["manzana", "mandarina", "pera",
"uva"]
```

```
# Ordenar una lista frutas.sort()  # Ordena la lista
alfabéticamente
print(frutas)  # Imprime ["manzana", "mandarina", "pera", "uva"]
```

Con las listas usamos estas funciones:

- len(): Retorna la longitud de una lista, es decir, la cantidad de elementos que contiene.
- append(): Agrega un elemento al final de la lista.
- insert(): Inserta un elemento en una posición específica de la lista.
- remove(): Elimina la primera aparición de un elemento específico de la lista.
- pop(): Elimina y retorna el último elemento de la lista, o un elemento en una posición específica si se proporciona un índice.
- index(): Retorna el índice de la primera aparición de un elemento específico en la lista.
- count(): Retorna la cantidad de veces que un elemento específico aparece en la lista.
- sort(): Ordena los elementos de la lista en orden ascendente.
- reverse(): Invierte el orden de los elementos en la lista.
- extend(): Combina una lista existente con otra lista o cualquier otro iterable, agregando sus elementos al final de la lista original.

2. 1.2. TUPLAS

En Python, una tupla es un tipo de dato que se utiliza para almacenar una colección ordenada e inmutable de elementos. A diferencia de las listas, las tuplas no pueden modificarse después de su creación. Son objetos estáticos cuyos elementos no pueden agregarse, eliminarse ni modificarse una vez que se han definido.

Para crear una tupla en Python, se utilizan paréntesis () y los elementos se separan por comas. Aquí tienes un ejemplo de cómo crear una tupla:

```
frutas = ("manzana", "banana", "naranja")
```

Las tuplas se utilizan principalmente en situaciones en las que se desea asegurar que los elementos no sean modificados después de su creación. Algunas razones comunes para utilizar tuplas son:

Asignación múltiple: Las tuplas se pueden utilizar para asignar múltiples valores a múltiples variables en una sola instrucción.

x, y, z = (10, 20, 30)

Intercambio de valores: Las tuplas se pueden utilizar para intercambiar los valores de dos variables sin necesidad de una variable temporal adicional.

```
a = 5 b = 10 a, b = b, a  # Intercambia los valores
de a y b
```

Retorno de múltiples valores en una función: Una función puede devolver una tupla que contiene varios valores, que luego pueden ser asignados a variables separadas.

```
def obtener_coordenadas():
    x = 10     y
= 20     return
x, y


coordenada_x, coordenada_y = obtener_coordenadas()
```

Utilización en estructuras de datos inmutables: Las tuplas se pueden utilizar como claves en diccionarios u otros elementos de estructuras de datos que requieren objetos inmutables.

```
punto = (5, 10) diccionario =
{punto: "valor"}
```

Recordá que, debido a que las tuplas son inmutables, no puedes agregar, eliminar o modificar elementos después de haber sido creadas. Sin embargo, puedes acceder a los elementos de una tupla utilizando índices, al igual que en las listas.

Las tuplas son útiles cuando se necesita almacenar una colección de valores que no cambiarán y se desea asegurar su integridad y evitar modificaciones accidentales.

Con las tuplas utilizamos las siguientes funciones:

- len(): Retorna la longitud de una tupla, es decir, la cantidad de elementos que contiene.
- index(): Retorna el índice de la primera aparición de un elemento específico en la tupla.
- count(): Retorna la cantidad de veces que un elemento específico aparece en la tupla.
- sorted(): Retorna una nueva tupla que contiene los elementos de la tupla original, ordenados en orden ascendente.
- max(): Retorna el valor máximo de una tupla, considerando el orden de los elementos.
- min(): Retorna el valor mínimo de una tupla, considerando el orden de los elementos.

3. 1.3. DICCIONARIOS

En Python, un diccionario es una estructura de datos que se utiliza para almacenar una colección de pares clave-valor. A diferencia de las listas y tuplas, los diccionarios no están ordenados y se accede a los valores utilizando sus claves en lugar de índices.

Un diccionario es útil cuando se necesita almacenar y recuperar información utilizando una clave única. Se utiliza en situaciones donde se requiere una búsqueda rápida y eficiente de valores asociados a una clave. Algunos casos comunes de uso de diccionarios incluyen:

- Almacenar datos relacionados: Un diccionario permite asociar valores a claves, lo que es útil para representar relaciones y estructuras complejas.
- Búsquedas eficientes: Los diccionarios permiten acceder a los valores rápidamente utilizando las claves como referencia, en lugar de tener que recorrer una lista o realizar operaciones complejas de búsqueda.
- Configuraciones y opciones: Los diccionarios son útiles para almacenar configuraciones y opciones en aplicaciones, ya que permiten acceder a los valores mediante nombres descriptivos.

Para crear un diccionario en Python, se utilizan llaves {} y se separan los pares clave-valor por comas. Acá tenés un ejemplo de cómo crear un diccionario:

```
persona = {
    "nombre": "Juan",
    "edad": 30,
    "ciudad": "Madrid"
}
```

Podés acceder a los valores de un diccionario utilizando sus claves:

```
print(persona["nombre"])  # Imprime "Juan"
print(persona["edad"])  # Imprime 30
```

También podés modificar los valores de un diccionario asignando nuevos valores a las claves:

```
persona["edad"] = 35 print(persona["edad"])  #
Imprime 35
```

Además, los diccionarios en Python tienen varios métodos incorporados que se pueden utilizar para realizar operaciones comunes, como agregar pares clavevalor, eliminar elementos, obtener listas de claves o valores, etc.

Acá tenés algunos ejemplos de uso de diccionarios y sus métodos:

```
# Agregar un par clave-valor persona["ocupacion"] =
"Ingeniero"

# Eliminar un elemento del diccionario del
persona["ciudad"]

# Obtener una lista de claves claves = persona.keys()
print(claves)  # Imprime ["nombre", "edad", "ocupacion"]

# Obtener una lista de valores valores =
persona.values() print(valores)  # Imprime ["Juan", 35,
"Ingeniero"]
```

Los diccionarios en Python son una poderosa herramienta para almacenar y organizar datos de manera eficiente utilizando claves y valores. Puedes explorar más funcionalidades y métodos de los diccionarios en la documentación oficial de Python.

Con los diccionarios usamos las siguientes funciones:

- len(): Retorna la cantidad de pares clave-valor en el diccionario.
- keys(): Retorna una vista iterable de las claves del diccionario.
- values(): Retorna una vista iterable de los valores del diccionario.
- items(): Retorna una vista iterable de los pares clave-valor del diccionario.
- get(): Retorna el valor asociado a una clave especificada. Si la clave no existe, retorna un valor predeterminado opcional en lugar de generar un error.
- update(): Actualiza el diccionario con pares clave-valor de otro diccionario o iterable.
- pop(): Elimina y retorna el valor asociado a una clave especificada. Si la clave no existe, se puede proporcionar un valor predeterminado opcional. • popitem(): Elimina y retorna un par clave-valor arbitrario del diccionario.
- clear(): Elimina todos los pares clave-valor del diccionario, dejándolo vacío.
- copy(): Retorna una copia superficial (shallow copy) del diccionario.

4. 1.4. CONJUNTOS

En Python, un conjunto (set) es una estructura de datos que se utiliza para almacenar una colección desordenada de elementos únicos. Los conjuntos se

utilizan cuando se desea almacenar elementos sin duplicados y no se requiere un orden específico.

Los conjuntos son útiles en varias situaciones, entre ellas:

- Eliminación de duplicados: Los conjuntos garantizan que no haya elementos duplicados, lo que facilita la eliminación de duplicados de una lista u otro iterable.
- Verificación de pertenencia: Los conjuntos ofrecen una búsqueda eficiente de elementos, lo que permite verificar rápidamente si un elemento está presente en el conjunto.
- Operaciones matemáticas de conjuntos: Los conjuntos en Python soportan operaciones como intersección, unión, diferencia y diferencia simétrica, lo que facilita las operaciones de conjuntos en matemáticas y lógica.

Para crear un conjunto en Python, se utilizan llaves {} o la función set(). Los elementos del conjunto se separan por comas. Ejemplo de cómo crear un conjunto:

```
frutas = {"manzana", "banana", "naranja"}
```

También podés crear un conjunto a partir de una lista o cualquier otro iterable utilizando la función set():

```
numeros = set([1, 2, 3, 4, 5])
```

Una vez creado un conjunto, puedes realizar varias operaciones y utilizar métodos para trabajar con él:

Agregar elementos: Utiliza el método add() para agregar un elemento al conjunto.

```
frutas.add("pera")
```

Eliminar elementos: Utiliza los métodos remove() o discard() para eliminar un elemento del conjunto. La diferencia entre ellos es que remove() genera un error si el elemento no está presente, mientras que discard() no genera ningún error.

```
frutas.remove("banana") frutas.discard("manzana")
```

Verificar pertenencia: Utilizá la palabra clave in para verificar si un elemento está presente en el conjunto.

```
if "naranja" in frutas:

    print("La naranja está en el conjunto de frutas.")
```

Operaciones de conjuntos: Los conjuntos en Python tienen varios métodos para realizar operaciones matemáticas, como unión, intersección, diferencia y diferencia simétrica. Estos métodos incluyen union(), intersection(), difference() y symmetric_difference().

```
frutas_2 = {"pera", "durazno", "manzana"}
```

```
union_frutas = frutas.union(frutas_2)
interseccion_frutas = frutas.intersection(frutas_2)
diferencia_frutas = frutas.difference(frutas_2)
diferencia_simetrica_frutas = frutas.symmetric_difference(frutas_2)
```

Recordá que los conjuntos no mantienen un orden específico de los elementos y no permiten elementos duplicados. Estas características los hacen útiles en muchas situaciones donde se necesita almacenar elementos únicos y realizar operaciones eficientes de conjuntos.

Con los conjuntos utilizamos las siguientes funciones: •

add(elemento): Agrega un elemento al conjunto.

- remove(elemento): Remueve un elemento específico del conjunto. Si el elemento no está presente, genera un error.
- discard(elemento): Remueve un elemento del conjunto si está presente. No genera un error si el elemento no existe.
- pop(): Remueve y retorna un elemento arbitrario del conjunto.
- clear(): Remueve todos los elementos del conjunto, dejándolo vacío.
- copy(): Retorna una copia superficial (shallow copy) del conjunto.
- len(): Retorna la cantidad de elementos en el conjunto.
- in: Verifica si un elemento está presente en el conjunto.
- union(conjunto): Retorna un nuevo conjunto que es la unión de dos conjuntos.
- intersection(conjunto): Retorna un nuevo conjunto que es la intersección de dos conjuntos.
- difference(conjunto): Retorna un nuevo conjunto que contiene los elementos del primer conjunto que no están en el segundo conjunto.
- symmetric_difference(conjunto): Retorna un nuevo conjunto que contiene los elementos que están en uno de los conjuntos, pero no en ambos.

5. 1.5. PILAS

En Python, una pila (stack) es una estructura de datos que sigue el principio de "último en entrar, primero en salir" (LIFO, por sus siglas en inglés Last-In, FirstOut). Se utiliza para almacenar elementos donde el último elemento agregado es el primero en ser retirado.

La pila se utiliza para resolver problemas que implican un seguimiento temporal o jerarquía de elementos, como el manejo de llamadas a funciones, el rastreo de operaciones en la recursión, la evaluación de expresiones postfix (notación polaca inversa), entre otros.

En Python, puedes implementar una pila utilizando una lista. Ejemplo de cómo crear una pila y realizar operaciones básicas en ella:

```
pila = []  # Crear una pila vacía
 pila.append(1)  # Agregar elementos a la pila
pila.append(2) pila.append(3) print(pila)  #
Imprimir la pila: [1, 2, 3]
elemento = pila.pop()  # Retirar el último elemento de la pila
print(elemento)  # Imprimir el elemento retirado: 3 print(pila)
# Imprimir la pila actualizada: [1, 2]
```

En el ejemplo anterior, utilizamos la función append() para agregar elementos a la pila y la función pop() para retirar el último elemento agregado. Al retirar un elemento, se elimina de la pila y se devuelve su valor. El resultado es una pila que se va actualizando a medida que se agregan y retiran elementos.

La principal razón para utilizar una pila es cuando necesitamos manejar elementos en el orden LIFO. Algunos casos de uso comunes de las pilas incluyen:

- Implementación de algoritmos como el recorrido en profundidad (DFS, por sus siglas en inglés Depth-First Search) en grafos y árboles.
- Evaluación de expresiones postfix (notación polaca inversa) en matemáticas y programación.
- Manejo de llamadas a funciones y rastreo de operaciones en la recursión. • Administración de la navegación y el historial de páginas en aplicaciones web.

Recuerda que en Python, las pilas se pueden implementar utilizando listas, pero también existen implementaciones más especializadas de pilas en el módulo collections de Python, como deque, que ofrecen una mejor eficiencia en ciertos casos.

6. 1.6. COLAS

En Python, una cola (queue) es una estructura de datos que sigue el principio de "primero en entrar, primero en salir" (FIFO, por sus siglas en inglés First-In, FirstOut). Se utiliza para almacenar elementos donde el primer elemento agregado es el primero en ser retirado.

La cola se utiliza en situaciones donde es importante mantener el orden de llegada de los elementos, como en la gestión de tareas, la planificación de procesos, el manejo de solicitudes en un servidor, etc.

En Python podemos implementar una cola utilizando la lista estándar y sus operaciones. Ejemplo de cómo crear una cola y realizar operaciones básicas en ella sin importar ningún módulo adicional:

```
cola = []  # Crear una cola vacía cola.append(1)
# Agregar elementos a la cola cola.append(2)
cola.append(3) print(cola)  # Imprimir la cola:
[1, 2, 3]
elemento = cola.pop(0)  # Obtener y retirar el primer elemento de la cola print(elemento)  # Imprimir el elemento obtenido: 1
print(cola)  # Imprimir la cola actualizada: [2, 3]
```

En este caso, utilizamos una lista (cola) para representar la cola. La función append() se utiliza para agregar elementos a la cola al final de la lista, y la función pop(0) se utiliza para obtener y retirar el primer elemento de la cola.

Sin embargo, es importante tener en cuenta que utilizar una lista estándar para implementar una cola puede no ser tan eficiente en términos de tiempo de ejecución, especialmente si la cola es grande y se realizan operaciones frecuentes de inserción y eliminación en la parte delantera de la lista. En esos casos, la clase Queue del módulo queue o una implementación especializada como deque del módulo collections proporcionan una mejor eficiencia y funcionalidad incorporada para trabajar con colas.

Por lo tanto, si necesitas una implementación más eficiente o requieres funcionalidades adicionales, se recomienda utilizar la clase Queue del módulo queue o deque del módulo collections.