Lectura

Estructuras de datos en Python: Operaciones con conjuntos y módulo ***Collections***

Python Intermedio

Introducción

En esta segunda unidad se abordan herramientas que amplían las posibilidades de trabajo con estructuras de datos en Python. Estas técnicas permiten organizar, procesar y analizar información de forma más clara y eficiente, aportando soluciones prácticas a problemas comunes de programación.

Dentro de los temas estudiados, destacan las **operaciones con conjuntos**, las cuales, facilitan tareas como: verificar pertenencia, comparar colecciones o combinar elementos sin duplicados, así como las estructuras avanzadas del módulo ***collections***. Cada una de ellas aporta funcionalidades específicas que optimizan la gestión de datos, evitando la necesidad de implementar soluciones manuales más complejas.

El aprendizaje y aplicación de estas herramientas, refuerzan la capacidad de escribir código de una manera más profesional, escalable y mantenible; características fundamentales en proyectos reales, donde la claridad y el rendimiento son aspectos clave.

ChainMap

ChainMap es una estructura del módulo *collections* que permite **agrupar múltiples diccionarios** en una sola vista lógica, **sin necesidad de fusionarlos**. Esto resulta muy útil cuando queremos realizar búsquedas en cascada, porque, al consultar una clave, Python revisa cada diccionario en orden, hasta encontrarla.

A continuación, se presenta el siguiente ejemplo:

El siguiente ejemplo **combina 2 diccionarios de configuración: los colores del usuario sobrescriben a los valores predeterminados**

- ➔ *ChainMap(config_usuario, config_default)* crea una vista conjunta.
- ➔ Cuando buscamos "tema", lo encuentra en *config_usuario*.
- ➔ Si la clave no está en el primer diccionario, la busca en *config_default*.
- ➔ Esto permite usar valores del usuario, sin perder los predeterminados.

```
from collections import ChainMap

config_default = {"tema": "claro", "idioma": "español"}
config_usuario = {"tema": "oscuro"}

config = ChainMap(config_usuario, config_default)

print(config["tema"])      # oscuro
print(config["idioma"])    # español
```

En el siguiente ejemplo **se gestionan diferentes ámbitos de variables, sin mezclarlos:**

```
from collections import ChainMap

globales = {"x": 1, "y": 2}
locales = {"y": 99, "z": 3}

ambiente = ChainMap(locales, globales)

print(ambiente["y"])  # 99 (local sobrescribe global)
print(ambiente["x"])  # 1  (solo está en globales)
```

- ➔ *locales* redefine “y”, sobrescribiendo el valor de *globales*.
- ➔ Cuando se consulta “*x*”, solo existe en el diccionario global.
- ➔ Esto ilustra cómo *ChainMap* permite **jerarquizar valores**.

Los casos de uso prácticos de *ChainMap* son:

- Fusionar configuraciones por niveles (default → usuario → sistema).

- Gestionar entornos variables en programas.
- Simplificar búsquedas jerárquicas en múltiples diccionarios.

Counter

Counter es una clase del módulo *collections*, que **permite contar la frecuencia de los elementos en un iterable.** Devuelve un diccionario especializado, donde las claves son los elementos, y los valores, son las veces en que aparecen.

A continuación, se presentan algunos ejemplos:

- En el siguiente ejemplo **se obtiene la frecuencia de letras, en una palabra:**

```
from collections import Counter

texto = "banana"
conteo = Counter(texto)

print(conteo)
# Counter({'a': 3, 'n': 2, 'b': 1})
```

➔ Cada carácter de la cadena, se convierte en clave del *Counter*.

➔ "a" aparece 3 veces, "n" 2 veces y "b" una vez.

- En el siguiente ejemplo **se listan los elementos más frecuentes en una colección.**

```
from collections import Counter

colores = ["rojo", "azul", "rojo", "verde", "azul", "rojo"]
conteo = Counter(colores)

print(conteo.most_common(2))
# [('rojo', 3), ('azul', 2)]
```

➔ *most_common(2)* devuelve una lista con las 2 claves más frecuentes.

➔ "rojo" aparece 3 veces y "azul" 2.

Los casos de uso prácticos de *Counter* son:

- Análisis de frecuencia en textos (palabras más usadas).
- Estadísticas rápidas en colecciones de datos.
- Identificación de elementos repetidos en listas.

DefaultDict

defaultdict es una subclase de *dict* en el módulo *collections*. La diferencia principal con un diccionario normal, es que asigna automáticamente un valor por defecto a las claves que no existen, evitando errores como *KeyError*.

A continuación, se presentan los siguientes ejemplos:

- En este ejemplo **se muestra la manera en que se organiza una lista de pares en un diccionario, agrupando automáticamente por clave:**

```
from collections import defaultdict

pares = [("rojo", 1), ("azul", 2), ("rojo", 3)]

d = defaultdict(list)
for k, v in pares:
    d[k].append(v)

print(d)
# defaultdict(<class 'list'>, {'rojo': [1, 3], 'azul': [2]})
```

➔ *defaultdict(list)* crea listas vacías automáticamente.

➔ Cada clave agrupa sus valores, sin necesidad de verificar si existe previamente.

- En el siguiente ejemplo **se asigna un valor por defecto para claves inexistentes usando *str*.**

```
1  from collections import defaultdict
2  d = defaultdict(str)
3  print(d["usuario"])
```

➔ *defaultdict(str)* crea automáticamente una cadena vacía "" cuando la clave no existe.

➔ Así, se evita un error y se puede trabajar directamente con el valor.

Los casos de uso prácticos de *defaultdict* son:

- Contadores acumulativos (con *int).*
- Agrupar valores sin inicializar listas manualmente.
- Inicializar estructuras complejas de forma automática.

Deque

deque (double-ended queue) es una estructura del módulo *collections* que **funciona como una cola de doble extremo**, lo que permite insertar y eliminar elementos de ambos lados con eficiencia **O(1)**. Es más rápido, en comparación con el uso de listas, cuando se trabaja con operaciones frecuentes en los extremos.

A continuación, se presentan algunos ejemplos:

- En el siguiente ejemplo **se utiliza una estructura FIFO (cola) o LIFO (pila):**

```python
from collections import deque

cola = deque(["a", "b", "c"])
cola.append("d")          # agrega al final
cola.appendleft("z")      # agrega al inicio
cola.pop()                # quita del final
cola.popleft()            # quita del inicio

print(cola)
# deque(['a', 'b', 'c'])
```

➔ Con *popleft()* se obtiene el primero en entrar (cola).

➔ Con *pop()* se obtiene el último en entrar (pila).

➔ *deque* permite emular ambas estructuras fácilmente.

- En el siguiente ejemplo **se crea una cola con un límite de tamaño:**

```python
from collections import deque

historial = deque(maxlen=3)
historial.extend([1, 2, 3])
historial.append(4)

print(historial)
# deque([2, 3, 4], maxlen=3)
```

➔ *maxlen=3* limita el tamaño máximo de la cola.

➔ Al insertar 4, se elimina automáticamente el primer elemento (1).

➔ Esto es útil para historiales o buffers.

NamedTuple

NamedTuple es una subclase de *tuple* del módulo *collections*, la cual permite acceder a los elementos de una tupla por nombre, además de hacerlo por índice, mejorando la legibilidad y claridad del código. Es inmutable, por lo que sus valores no se pueden modificar después de creados.

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo **se define una tupla con nombres de campo y se accede a sus valores por nombre, en lugar de índice:**

```
1   from collections import namedtuple
2
3   Persona = namedtuple("Persona", ["nombre", "edad"])
4   p = Persona(nombre="Ana", edad=25)
5
6   print(p.nombre)  # Ana
7   print(p[1])      # 25
```

➔ Se define Persona con los campos "nombre" y "edad".

➔ Se puede acceder tanto con p.nombre como con p[1].

➔ Esto da más claridad que usar sólo índices.

Los casos de uso prácticos de *NamedTuple* son:

- Representar registros de datos de forma clara (personas, vehículos, productos).
- Estructuras inmutables fáciles de leer y documentar.
- Situaciones donde se necesita algo más legible que una tupla, pero más ligero que una clase.

OrderedDict

OrderedDict es una subclase de *dict* del módulo *collections* que **mantiene el orden en que se insertan los elementos**. Aunque desde Python 3.7 los diccionarios estándar también conservan el orden, *OrderedDict* sigue siendo útil porque **ofrece métodos adicionales para manipular ese orden.**

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo **se muestra cómo cambiar la posición de una clave específica:**

```
from collections import OrderedDict

datos = OrderedDict([("a", 1), ("b", 2), ("c", 3)])
datos.move_to_end("a")

print(datos)
# OrderedDict([('b', 2), ('c', 3), ('a', 1)])
```

➔ *move_to_end("a")* mueve la clave "a" al final del diccionario.

➔ También se puede pasar *last=False* para moverla al inicio.

Los casos de uso prácticos de *OrderedDict* son:

- Mantener el orden de claves en reportes o exportaciones.
- Reorganizar datos dinámicamente según criterios específicos.
- Comparar estructuras de datos donde el orden es relevante.

Set

Un *set* en Python **es una estructura de datos que almacena elementos únicos y no ordenados.** Es muy eficiente para comprobar pertenencia y realizar operaciones matemáticas como unión, intersección o diferencia.

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo se muestra **cómo comprobar rápidamente si un elemento está en un conjunto:**

```python
frutas = {"manzana", "pera", "uva"}
print("pera" in frutas)   # True
print("sandía" in frutas) # False
```

➔ La operación in es muy rápida en conjuntos.

➔ Devuelve *True* si el elemento existe y *False* si no.

Los casos de uso prácticos de *set* son:

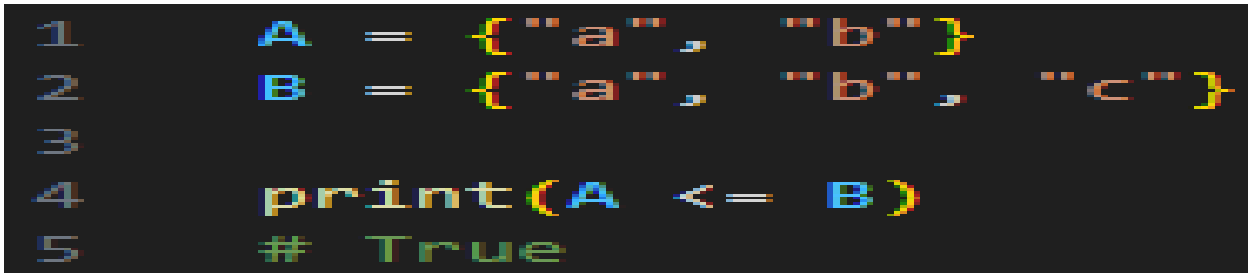
- Eliminar duplicados en listas de forma automática.
- Comprobar si un valor pertenece a un grupo de elementos.
- Resolver problemas de teoría de conjuntos en datos.

Subconjunto

Un conjunto A es subconjunto de otro conjunto B si todos los elementos de A están contenidos en B. En Python se puede comprobar con el método ***issubset()*** o con el operador **<=**.

A continuación, se presentan algunos ejemplos:

- En el siguiente ejemplo **se muestra cómo obtener el mismo resultado de manera más compacta:**



➔ A <= B significa “A es subconjunto de B”.

➔ Como "a" y "b" están dentro de B, el resultado es *True*.

- En el siguiente ejemplo **se muestra cómo distinguir entre subconjunto y subconjunto propio:**

➔ <= permite igualdad, por eso X <= Y es *True*.

➔ < exige que X tenga menos elementos, por eso X < Y es *False*.

Los casos de uso prácticos de subconjuntos son:

- Validar permisos.
- Comprobar si una lista de requisitos está cubierta por otra lista mayor.
- Comparar relaciones de inclusión entre grupos de datos.

Superconjunto

Un **superconjunto** es un conjunto que **incluye todos los elementos de otro conjunto**, pudiendo contener otros elementos adicionales. Se utiliza para comprobar si una colección de datos es tan amplia como para abarcar otra. En Python, esta relación se verifica mediante el método ***issuperset()*** o con el operador **>=**.

A continuación, se presentan algunos ejemplos:

- El siguiente ejemplo **muestra cómo aplicar la misma lógica de forma más compacta:**

```
X = {"rojo", "azul", "verde"}
Y = {"rojo", "azul"}

print(X >= Y)
# True
```

➔ X >= Y significa “X es superconjunto de Y”.

➔ Como "rojo" y "azul" están dentro de X, la condición se cumple.

- En el siguiente ejemplo **se muestra cómo diferenciar entre un superconjunto y un superconjunto propio:**

```
M = {1, 2, 3}
N = {1, 2, 3}

print(M > N)    # False
print(M >= N)   # True
```

➔ >= permite igualdad, por eso M >= N es *True*.

➔ > exige que M tenga más elementos que N, por eso M > N es *False*.

Los casos de uso práctico para los superconjuntos son:

- Verificar que una colección de datos incluya a otra completamente.
- Comprobar que un conjunto de permisos contiene a los de otro rol.
- Comparar listas de inventario o catálogos completos, contra otros parciales.

Conjuntos disjuntos

Dos conjuntos son **disjuntos** cuando no comparten ningún elemento en común. En Python, esto se puede verificar con el método ***isdisjoint()***, que devuelve *True* si no hay elementos repetidos entre ambos conjuntos.

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo **se valida si una lista de productos disponibles no coincide con una lista de prohibidos:**

```
1   permitidos = {"manzana", "pera", "uva"}
2   prohibidos = {"fresa", "sandía"}
3
4   print(permitidos.isdisjoint(prohibidos))  # True
```

➔ permitidos y prohibidos no comparten ningún elemento.

➔ El resultado *True* confirma que la lista de productos puede usarse sin problema.

Los casos de uso prácticos de conjuntos disjuntos son:

- Comprobar que dos colecciones de datos no tienen solapamiento.
- Validar listas de valores permitidos/prohibidos.
- Simplificar operaciones lógicas en clasificación de conjuntos.

Diferencia (conjuntos)

La diferencia entre conjuntos devuelve los elementos que están en el primer conjunto, pero no en el segundo. En Python se usa el operador - o el método ***difference()***.

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo **se aplica la misma lógica, pero utilizando el método *difference()*:**

```
1   frutas = {"manzana", "pera", "uva"}
2   citricos = {"naranja", "limón", "pera"}
3
4   print(frutas.difference(citricos))
5   # {'uva', 'manzana'}
```

➔ *difference()* revisa los elementos de frutas y elimina los que aparecen en citricos.

➔ pera está en ambos, así que no aparece en el resultado.

Nota: Se puede aplicar lo mismo a más de 2 conjuntos al mismo tiempo. ¡Inténtalo!

Los casos de uso prácticos de la diferencia de conjuntos son:

- Comparar dos colecciones de datos para ver qué falta en una lista respecto a otra.

- Detectar registros exclusivos en un dataset.
- Filtrar elementos de una lista eliminando duplicados que existen en otra.

Diferencia simétrica (conjuntos)

La diferencia simétrica entre dos conjuntos devuelve los elementos que están en uno u otro, pero no en ambos. En Python se usa el operador **^** o el método ***symmetric_difference()***.

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo **se obtiene el mismo resultado, pero utilizando el método *symmetric_difference()*.**

```
frutas = {"manzana", "pera", "uva"}
citricos = {"naranja", "limón", "pera"}

print(frutas.symmetric_difference(citricos))
# {'manzana', 'uva', 'naranja', 'limón'}
```

➔ El elemento común "pera" se elimina.

➔ El resultado contiene solo los elementos exclusivos de cada conjunto.

Nota: Puedes aplicar lo mismo a más de 2 conjuntos, pero esto sucederá de forma secuencial. ¡Inténtalo!

Los casos de uso prácticos de la diferencia simétrica de conjuntos son:

- Detectar diferencias exclusivas entre dos listas de datos.
- Comparar inventarios para saber qué productos pertenecen solo a un conjunto.
- Identificar cambios entre versiones de un conjunto de registros.

Intersección

La intersección entre conjuntos devuelve los elementos que aparecen en todos los conjuntos involucrados. En Python se puede usar el operador "**&**" o el método ***intersection()***.

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo **se muestra la misma lógica, pero aplicando el método *intersection()*:**

```
frutas = {"manzana", "pera", "uva"}
citricos = {"limón", "naranja", "pera"}

print(frutas.intersection(citricos))
# {'pera'}
```

➔ *intersection()* compara ambos conjuntos.

➔ "pera" es el único elemento presente en los dos, así que queda en el resultado.

Nota: Puedes aplicar la misma lógica a más de 2 conjuntos para obtener elementos comunes.

Los casos de uso prácticos de intersección son:

- Detectar coincidencias entre listas (ej. estudiantes que están en dos cursos).
- Encontrar registros duplicados en diferentes bases de datos.
- Extraer valores comunes en análisis de datos.

Unión

La unión de conjuntos devuelve un nuevo conjunto con todos los elementos únicos, presentes al menos en uno de los conjuntos. En Python se puede realizar con el operador "**|**" o con el método ***union()***.

A continuación, se presenta el siguiente ejemplo:

- En el siguiente ejemplo **se muestra cómo aplicar la misma lógica, aplicando el método *unión()*:**

```
frutas = {"manzana", "pera"}
citricos = {"naranja", "limón"}

print(frutas.union(citricos))
# {'manzana', 'pera', 'naranja', 'limón'}
```

➔ *union()* combina todos los elementos de frutas y citricos.

➔ El resultado incluye todas las frutas sin duplicados.

NOTA: Es posible aplicar la misma lógica a más de 2 conjuntos.

Los casos de uso práctico de unión son:

- Fusionar datos de varias fuentes sin duplicados.
- Combinar listas de usuarios, productos o registros.
- Crear colecciones globales a partir de subconjuntos.

Update()

El método *update()* se utiliza en conjuntos (*set*) para agregar múltiples elementos de otro iterable al conjunto original. Los duplicados se eliminan automáticamente, ya que un conjunto solo guarda valores únicos.

A continuación, se presentan algunos ejemplos:

- En el siguiente ejemplo se muestra **cómo fusionar 2 conjuntos directamente en uno solo:**

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

A.update(B)
print(A)
# {1, 2, 3, 4, 5}
```

➔ A.update(B) agrega todos los elementos de B dentro de A.

➔ Como 3 ya estaba en A, no se repite.

➔ El resultado es {1, 2, 3, 4, 5}.

- En el siguiente ejemplo se muestra **cómo agregar cada carácter de una cadena, como elemento de un conjunto:**

```
letras = {"a", "b"}
letras.update("cde")

print(letras)
# {'a', 'b', 'c', 'd', 'e'}
```

➔ Una cadena es un iterable, por lo que cada letra se agrega por separado.

➔ El conjunto final contiene todos los caracteres únicos.

Los casos de uso prácticos de *update()* son:

- Fusionar datos de diferentes fuentes en un solo conjunto.
- Agregar múltiples valores de una lista sin necesidad de usar un bucle.
- Garantizar que no haya duplicados al ampliar un conjunto.

Conclusión

En esta unidad revisamos herramientas avanzadas para el manejo de estructuras de datos en Python. A través de operaciones con conjuntos, se reforzó la capacidad de analizar, comparar y organizar colecciones de información de manera clara y precisa.

Asimismo, se exploraron estructuras especializadas del módulo *collections*, las cuales permiten resolver problemas comunes de conteo, agrupación, organización y combinación de datos con mayor eficiencia y menos esfuerzo de programación.

El dominio de estas herramientas no solo mejora la legibilidad y optimización del código, sino que también acerca al estudiante a un estilo de programación más profesional y escalable. Estas bases serán de gran utilidad en la siguiente unidad, enfocada en la **programación orientada a objetos**, donde la estructuración y reutilización del código cobran aún mayor relevancia.

Elaboró: Enrique Quezada Próspero
Contenido: Enrique Quezada Próspero
DI: Génesis Equihua

Referencias:

1. Campesato, O. (2023). *Intermediate Python*. Mercury Learning and Information.
2. Lutz, M. (2013). *Learning Python* (5th ed.). O'Reilly Media.
3. Python Software Foundation. (2025a). *Built-in types — set*. Recuperado el 11 de agosto de 2025, de: https://docs.python.org/3/library/stdtypes.html#set-types-set-frozenset
4. Python Software Foundation. (2025b). *collections — Container datatypes*. Recuperado el 11 de agosto de 2025, de: https://docs.python.org/3/library/collections.html
5. Python Software Foundation. (2025c). *Data Structures — Python 3.11.4 documentation*. Recuperado el 11 de agosto de 2025, de: https://docs.python.org/3/tutorial/datastructures.html