Recibe una cálida:

¡Bienvenida!

Te estábamos esperando 😁

alkemy

Clase grabada

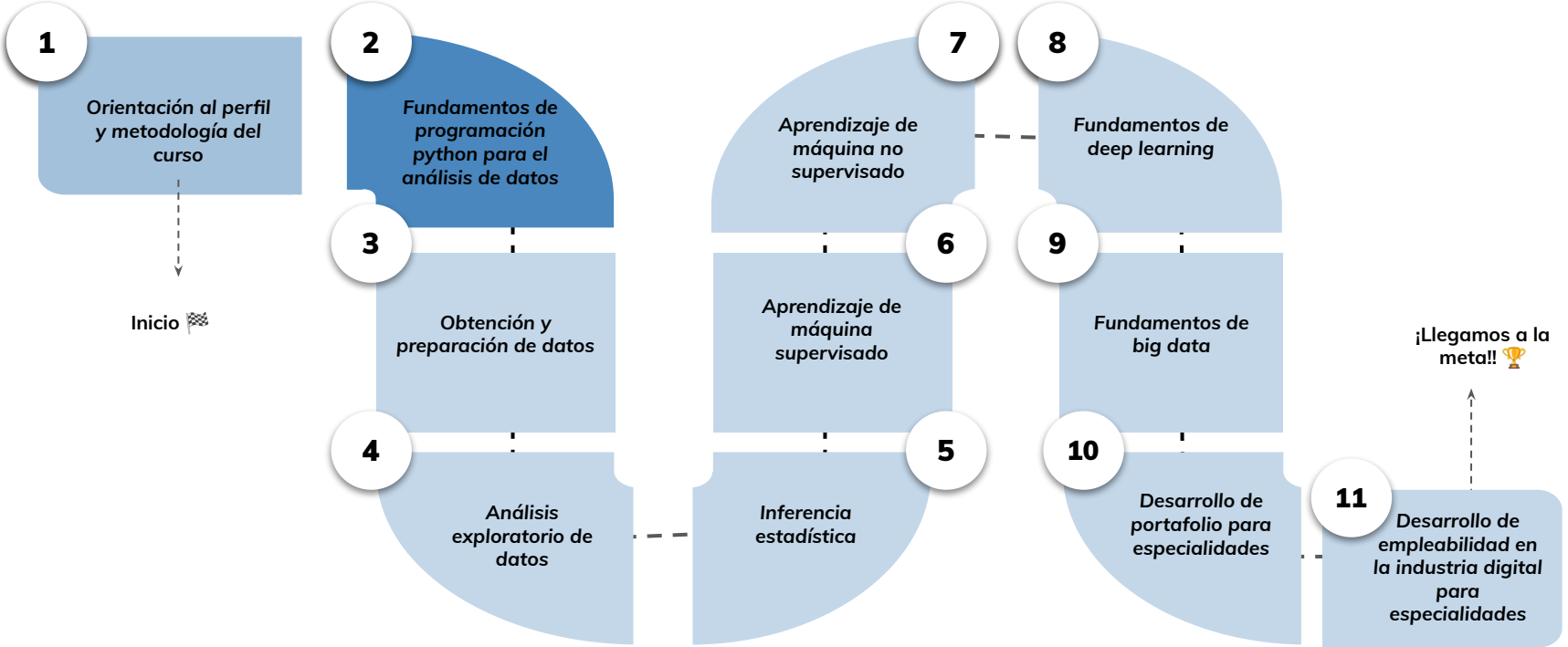
› Estructuras de dato en Python - Parte II

Aprendizaje Esperado 5: Aplicar las estructuras de dato de tipo colección del lenguaje python junto a sus características y utilidad para resolver un problema.

alkemy

Hoja de ruta

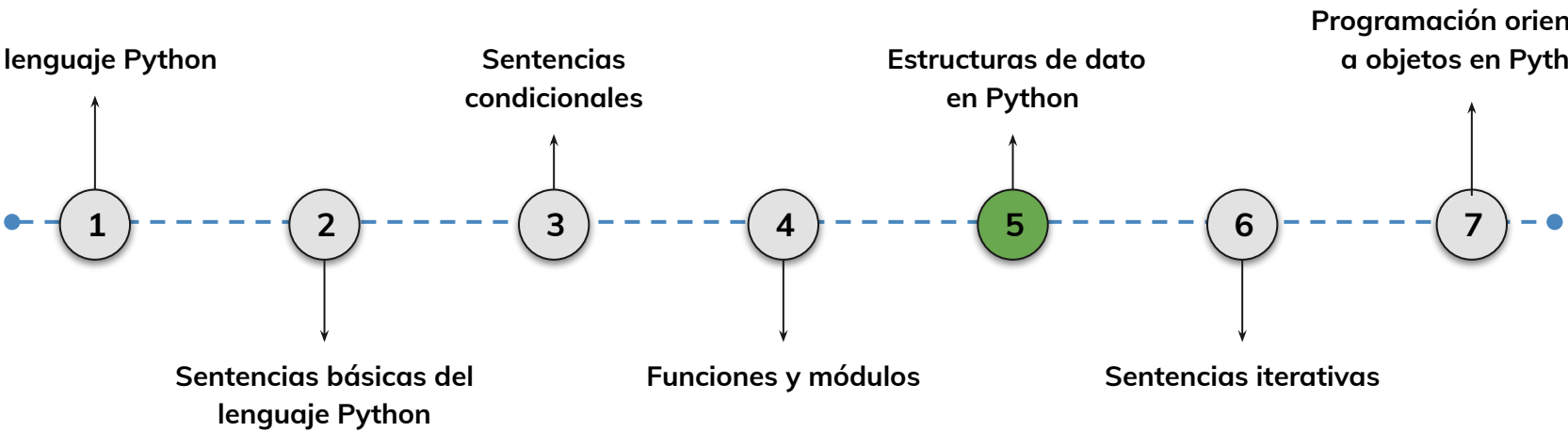
¿Cuáles skills conforman el programa? **Fundamentos de Ciencia de Datos**



alkemy

Roadmap de lecciones

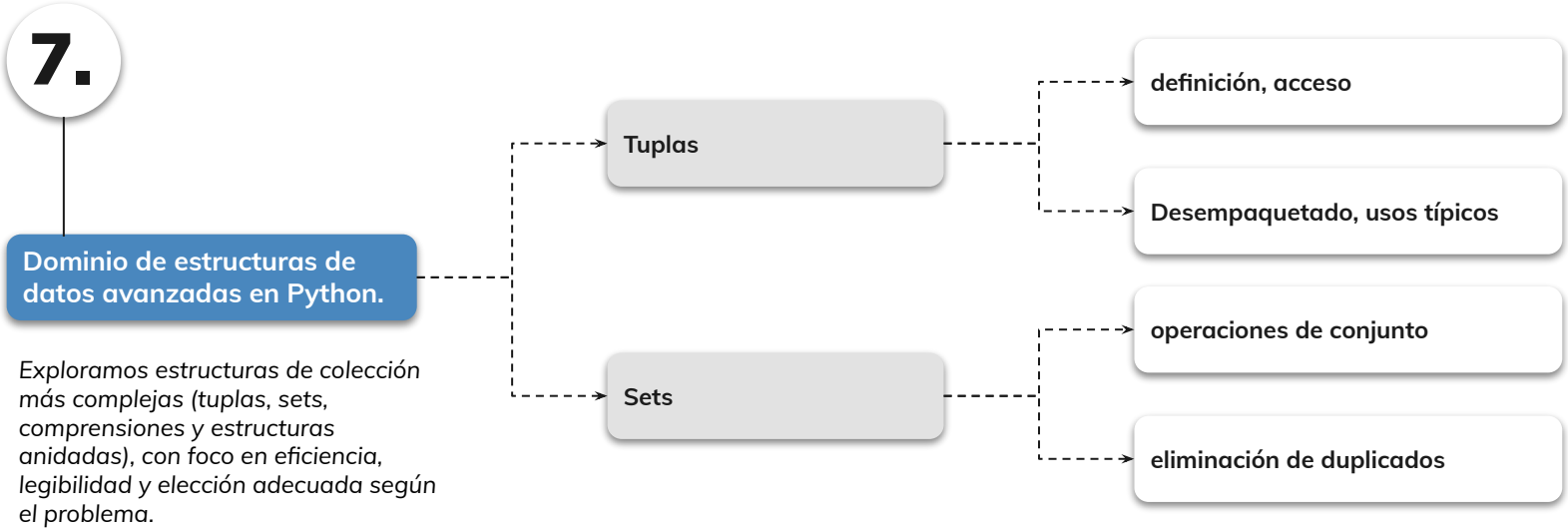
*¿Cuáles **lecciones** estaremos estudiando en este módulo?*



alkemy

Learning Path

¿Cuáles temas trabajaremos hoy?



alkemy

Objetivos de aprendizaje

¿Qué aprenderás?

- Comprender las características de tuplas y sets en Python
- Aplicar operaciones de conjunto como unión, intersección y diferencia
- Construir colecciones eficientes usando comprensiones
- Manipular estructuras anidadas (listas de diccionarios, diccionarios con listas, etc.)
- Seleccionar la estructura de datos adecuada según el tipo de problema

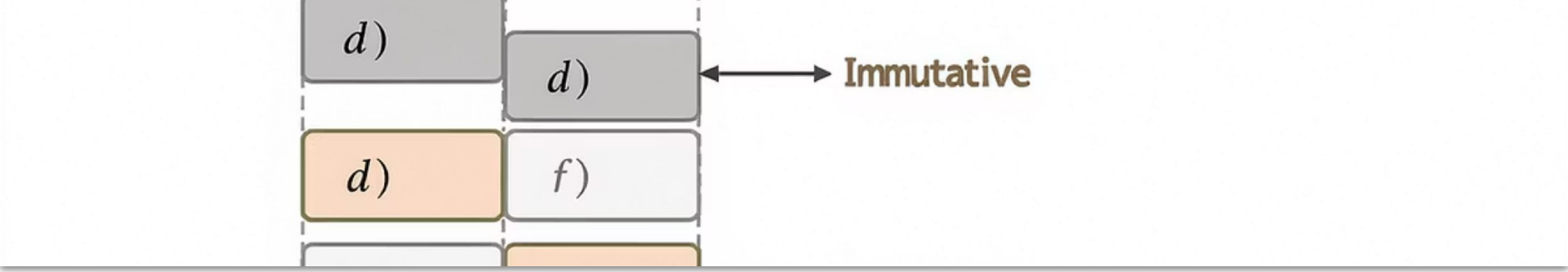
Repaso clase anterior

¿Quedó alguna duda?

En la clase anterior trabajamos 📚:

- → *Usamos listas y diccionarios para representar información de estudiantes*
- → *Aplicamos métodos como append(), sort(), get() y for para mostrar o actualizar datos*
- → *Creamos rutinas básicas para calcular promedios y ordenar colecciones*
- → *Entendimos cómo combinar estructuras (listas de diccionarios) para representar sistemas simples*

Estructuras de dato en python



Tuplas en Python

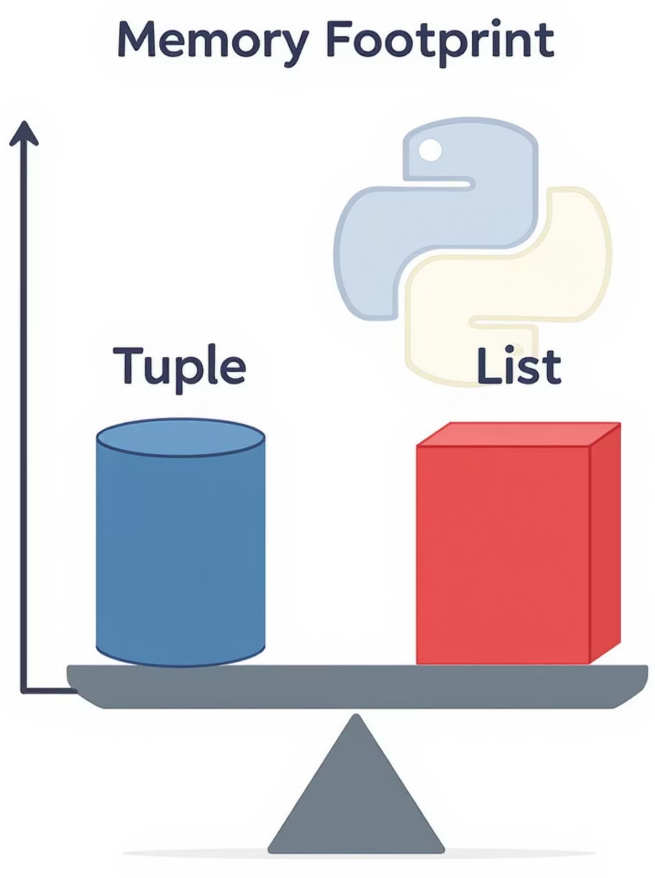
Características de una Tupla

Las tuplas son estructuras de datos ordenadas e inmutables, lo que significa que no se pueden modificar después de su creación. Las tuplas son útiles para almacenar datos constantes que no deben cambiar, como coordenadas, y son más rápidas que las listas en términos de acceso.

< >

Ventajas de las tuplas

Las tuplas son más eficientes en términos de memoria y rendimiento que las listas, lo que las convierte en una opción eficiente cuando se necesita trabajar con conjuntos de datos fijos.



Creación de una Tupla

Para crear una tupla, se utilizan paréntesis () y los elementos se separan por comas. Las tuplas también pueden crearse sin paréntesis, usando solo comas.

```
mi_tupla = (10, 20, "Python")
mi_tupla_sin_parentesis = 1, 2, 3
print(mi_tupla)  # Salida: (10, 20, 'Python')
print(mi_tupla_sin_parentesis)  # Salida: (1, 2, 3)
```

< >

Acceso a Elementos en una Tupla

Se puede acceder a los elementos de una tupla mediante su índice, de la misma manera que en las listas.

```
print(mi_tupla[1])  # Salida: 20
```

Empaquetado y Desempaquetado de Tuplas

El empaquetado permite agrupar múltiples valores en una tupla, mientras que el desempaquetado extrae estos valores en variables individuales.

```
coordenadas = (4, 5)
x, y = coordenadas
print(x)  # Salida: 4
print(y)  # Salida: 5
```

< >

Usos de las Tuplas en Programación

Datos inmutables

Perfectas para representar fechas, coordenadas y otros datos que no deben cambiar

Retorno múltiple

Ideales para funciones que devuelven múltiples valores en un solo retorno

Claves de diccionario

Pueden usarse como claves en diccionarios debido a su inmutabilidad

alkemy

< >

Operaciones con tuplas

Concatenación

Unir tuplas con el operador +

Repetición

Repetir elementos con el operador *

Slicing

Extraer subtuplas con la notación de rebanado

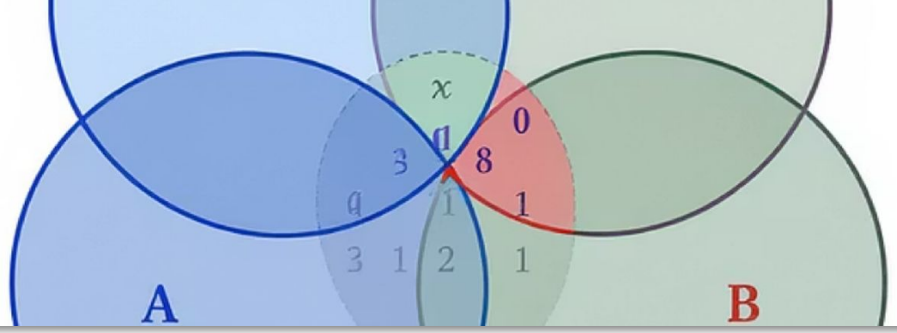
Conversión

Convertir entre tuplas y listas con tuple() y list()

Sets en Python

Características de un Set

Un set es una estructura de datos no ordenada que no permite elementos duplicados, lo cual lo convierte en una excelente opción para almacenar colecciones de datos únicos. Los sets son mutables, lo que significa que sus elementos se pueden añadir o eliminar, pero no permiten el acceso por índice debido a su naturaleza desordenada.



Ventajas de los sets

Los sets son útiles para operaciones de conjunto, como la unión, intersección y diferencia. Su capacidad para eliminar duplicados automáticamente los hace ideales para limpiar datos.

Creación de un Set

Para crear un set, se utilizan llaves {} o la función set() para evitar crear un diccionario.

```
mi_set = {1, 2, 3, 4, 4, 5}
print(mi_set)  # Salida: {1, 2, 3, 4, 5} (elimina duplicados)
```

Agregar y Eliminar Elementos en un Set

Para agregar un elemento a un set, se utiliza el método add(). También se pueden eliminar elementos usando remove() o discard().

```
mi_set.add(6)
mi_set.discard(2)
print(mi_set)  # Salida: {1, 3, 4, 5, 6}
```

Operaciones de Conjunto con Sets

Los sets permiten realizar operaciones de conjuntos como la unión (|), intersección (&), y diferencia (-), útiles para manejar datos únicos en diferentes colecciones.

```
set_a = {1, 2, 3}
set_b = {3, 4, 5}

print(set_a | set_b)  # Unión: {1, 2, 3, 4, 5}
print(set_a & set_b)  # Intersección: {3}
print(set_a - set_b)  # Diferencia: {1, 2}
```

Eliminación de Elementos Duplicados con Sets

Los sets se utilizan comúnmente para eliminar duplicados en una lista de datos.

```
lista = [1, 2, 2, 3, 4, 4, 5]
mi_set = set(lista)
print(list(mi_set))  # Salida: [1, 2, 3, 4, 5]
```

< >

Operaciones adicionales con sets

Diferencia simétrica

Elementos que están en uno u otro set, pero no en ambos (^)

Superconjunto

Verificar si un set contiene a otro (>=)

Subconjunto

Verificar si un set es subconjunto de otro (<=)

Conjuntos disjuntos

Verificar si dos sets no tienen elementos en común (isdisjoint())

< >

Casos de uso de sets

Eliminación de duplicados

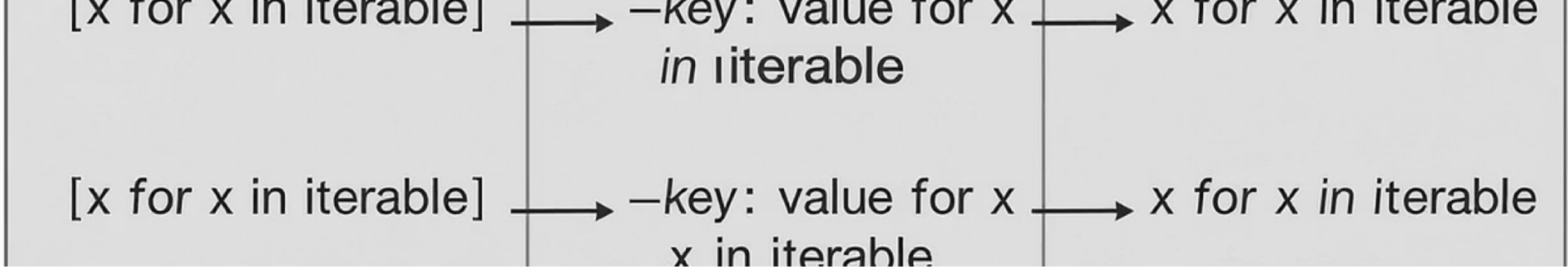
Convertir colecciones a sets para obtener valores únicos

Comparación de colecciones

Encontrar elementos comunes o diferentes entre conjuntos de datos

Verificación de pertenencia

Comprobar rápidamente si un elemento existe en una colección



Compresión de Listas, Diccionarios y Sets

Las comprensiones son técnicas avanzadas que permiten crear estructuras de datos de forma concisa y eficiente en una sola línea de código.

Compresión de Listas

La compresión de listas permite crear listas aplicando una operación o filtro a cada elemento de una colección existente en una sola línea de código, mejorando la eficiencia y la legibilidad.

```
numeros = [1, 2, 3, 4, 5]
cuadrados = [x**2 for x in numeros]
print(cuadrados)  # Salida: [1, 4, 9, 16, 25]
```

< >

Ventajas de la compresión de listas

Código más conciso

Reduce múltiples líneas a una sola expresión

Mayor legibilidad

Expresa claramente la intención de transformar o filtrar datos

Mejor rendimiento

Generalmente más rápido que los bucles tradicionales

Compresión de Diccionarios

La compresión de diccionarios es similar a la de listas, permitiendo aplicar una operación o filtro a los pares clave-valor en una sola línea.

```
diccionario = {x: x**2 for x in range(5)}
print(diccionario)  # Salida: {0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

< > Ejemplos de compresión de diccionarios

```
# 1. Crear un diccionario de cuadrados
cuadrados = {x: x**2 for x in range(10)}
print("Diccionario de cuadrados:", cuadrados)

# 2. Filtrar solo los valores pares de un diccionario original
original = {'a': 1, 'b': 2, 'c': 3, 'd': 4}
pares = {k: v for k, v in original.items() if v % 2 == 0}
print("Diccionario con valores pares:", pares)

# 3. Convertir los valores de un diccionario a mayúsculas
nombres = {'id1': 'ana', 'id2': 'luis', 'id3': 'maría'}
mayusculas = {k: v.upper() for k, v in nombres.items()}
print("Diccionario con valores en mayúsculas:", mayusculas)
```

Compresión de Sets

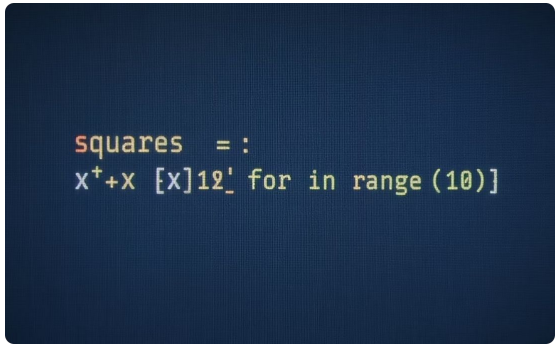
La compresión de sets elimina automáticamente duplicados y permite crear sets compactos y eficientes.

```
set_comp = {x for x in range(10) if x % 2 == 0}
print(set_comp)  # Salida: {0, 2, 4, 6, 8}
```

< >Ejemplos de compresión de sets

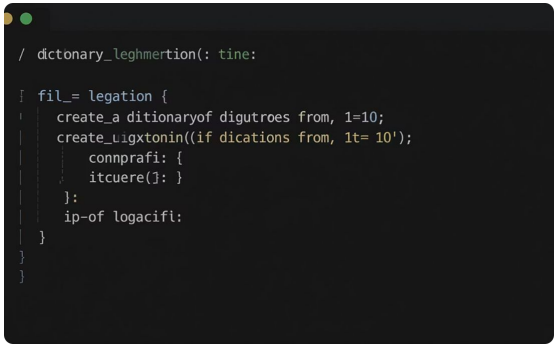
```
# 1. Crear un set de cuadrados del 0 al 9
cuadrados = {x**2 for x in range(10)}
print("Set de cuadrados:", cuadrados)

# 2. Filtrar valores pares de una colección
numeros = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
pares = {x for x in numeros if x % 2 == 0}
print("Set de números pares:", pares)

# 3. Extraer primeras letras de una lista de palabras
palabras = ["Hola", "Python", "Set", "Compresión"]
primeras_letras = {palabra[0] for palabra in palabras}
print("Primeras letras:", primeras_letras)
```

Comparación de comprensiones



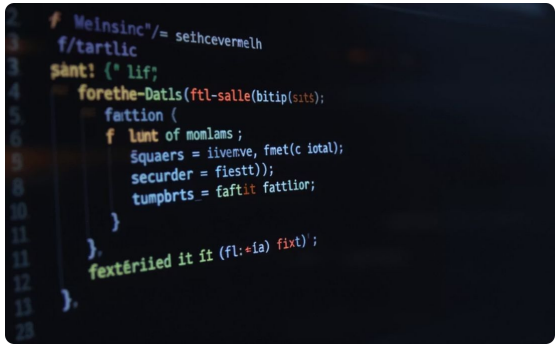
Compresión de listas

[expresión for elemento in iterable if condición]



Compresión de diccionarios

{clave: valor for elemento in iterable if condición}



Compresión de sets

{expresión for elemento in iterable if condición}

< >

Cuándo usar cada estructura

¿Necesitas mantener un orden?

Si necesitas mantener el orden de los elementos, considera usar listas o tuplas

¿Los datos deben ser inmutables?

Si los datos no deben cambiar, las tuplas son la mejor opción

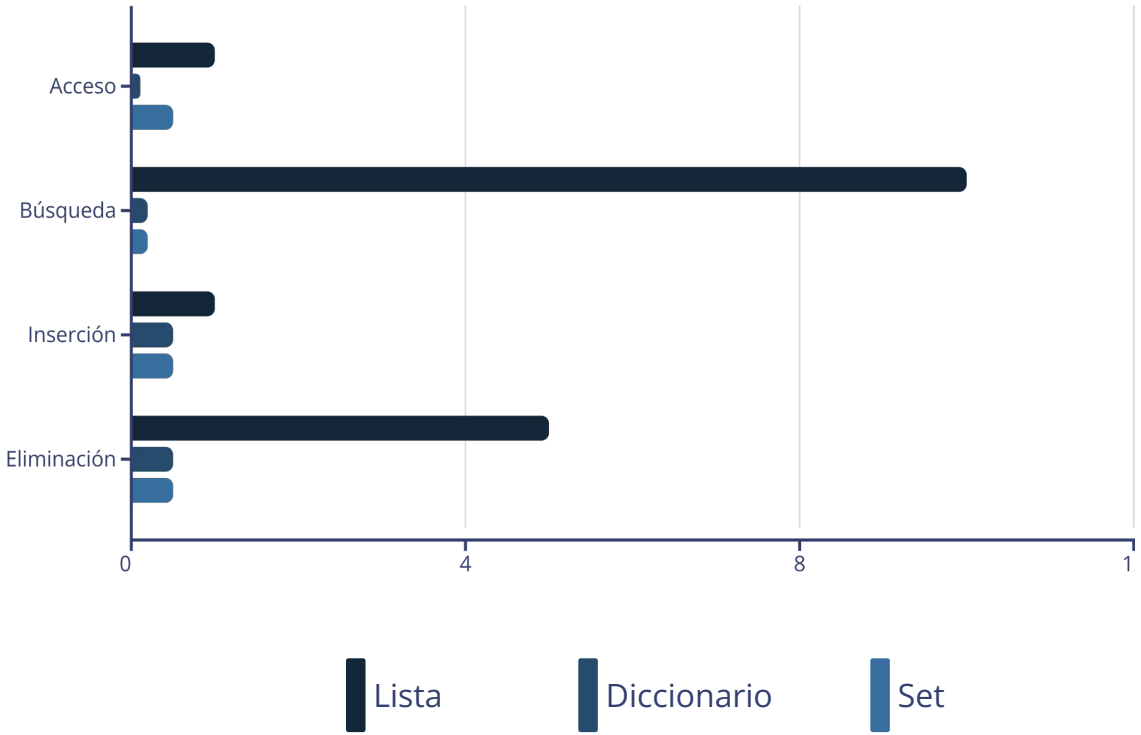
¿Necesitas acceso por clave?

Si necesitas acceder a los datos mediante una clave, usa diccionarios

¿Necesitas valores únicos?

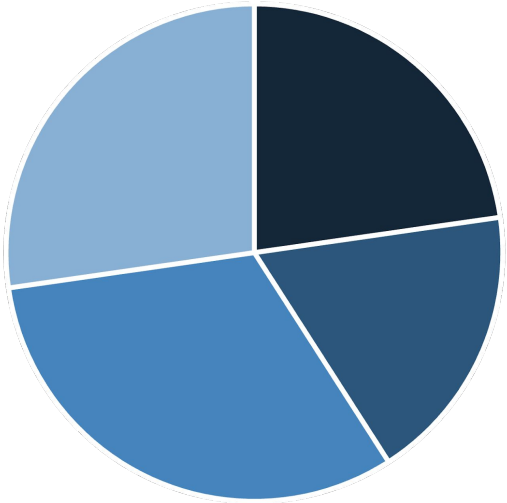
Si necesitas eliminar duplicados o realizar operaciones de conjunto, usa sets

< > Rendimiento de las estructuras



Tiempo relativo para operaciones comunes (valores más bajos son mejores)

< > Uso de memoria



Uso de memoria relativo para las diferentes estructuras de datos (valores más bajos son mejores)

< >

Estructuras anidadas

Listas de listas

Útiles para representar matrices y datos tabulares

Diccionarios anidados

Ideales para representar datos jerárquicos como JSON

Estructuras mixtas

Combinaciones de diferentes estructuras para casos complejos

< > Ejemplo de estructura mixta

```
# Estructura de datos para un sistema de
gestión de estudiantes

estudiantes = [
    {
        "nombre": "Ana",
        "edad": 22,
        "cursos": ["Python",
"Estadística", "Machine Learning"],
        "calificaciones": {
            "Python": 95,
            "Estadística": 88,
            "Machine Learning": 92
        }
    },
    {
        "nombre": "Carlos",
        "edad": 24,
        "cursos": ["Python", "Bases de Datos",
"Web"],
        "calificaciones": {
            "Python": 85,
            "Bases de Datos": 90,
            "Web": 78
        }
    }
]

# Ejemplo de acceso a datos
print("Nombre del primer estudiante:",
estudiantes[0]["nombre"])
print("Nota de Carlos en Web:",
estudiantes[1]["calificaciones"]["Web"])
```

< >

Buenas prácticas

Elegir la estructura adecuada

Seleccionar la estructura que mejor se adapte a las necesidades específicas del problema

Usar comprensiones con moderación

Las comprensiones mejoran la legibilidad solo cuando son simples y claras

Considerar el rendimiento

Tener en cuenta el rendimiento de las operaciones más frecuentes al elegir una estructura

Documentar estructuras complejas

Añadir comentarios que expliquen la estructura y propósito de los datos complejos

Recursos adicionales

Documentación oficial

La documentación de Python proporciona información detallada sobre todas las estructuras de datos

Tutoriales interactivos

Plataformas como Codecademy y DataCamp ofrecen cursos prácticos sobre estructuras de datos

Comunidad

Foros como Stack Overflow y Reddit tienen comunidades activas de Python donde puedes resolver dudas

< >

Ejercicios prácticos

1 **Manipulación de listas**

Crear una lista de números, filtrar los pares, y calcular su suma

2 **Operaciones con diccionarios**

Crear un diccionario de frecuencias de palabras en un texto

3 **Uso de sets**

Encontrar elementos comunes y diferentes entre dos listas

4 **Comprensiones**

Reescribir bucles tradicionales usando comprensiones de listas, diccionarios y sets

< >

Resumen

Listas

Colecciones ordenadas y mutables

Compresiones

Sintaxis concisa para crear y transformar colecciones

Diccionarios

Pares clave-valor para búsquedas rápidas

Tuplas

Colecciones ordenadas e inmutables

Sets

Colecciones desordenadas de elementos únicos

< >

Cierre

Las estructuras de datos en Python son herramientas poderosas que permiten almacenar y manipular datos de manera eficiente. Al comprender las características y el uso adecuado de listas, diccionarios, tuplas y sets, los programadores pueden seleccionar la estructura más adecuada para cada situación, optimizando el rendimiento y organizando el código de manera más clara y modular. La compresión de listas, diccionarios y sets facilita la creación de colecciones derivadas en un solo paso, mejorando la eficiencia del código.

Con el dominio de estas estructuras de datos, los programadores pueden enfrentar desafíos complejos en el manejo de datos y desarrollar aplicaciones avanzadas y escalables. Practicar el uso de estas estructuras y técnicas de compresión es clave para mejorar en Python y para optimizar el procesamiento de datos en proyectos de cualquier escala.

Live Coding

¿En qué consistirá la Demo?

Durante la demo trabajaremos con tuplas y sets para organizar datos inmutables y eliminar duplicados. Además, implementaremos comprensiones para simplificar transformaciones y estructuras anidadas para representar sistemas más complejos.

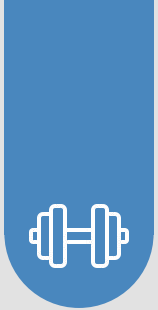
1. *Crear tuplas para representar coordenadas y retorno múltiple*
2. *Usar sets para encontrar elementos únicos y diferencias entre listas*
3. *Crear una lista con comprensión ([x for x in datos if x > 5])*
4. *Armar un diccionario con comprensión filtrando por valor*
5. *Construir una estructura de alumnos con cursos y calificaciones anidadas*

Tiempo: 25 Minutos

Momento:

Time-out!

⏰5 -10 min.



Ejercicio N° 1

Colecciones eficientes para datos complejos

Colecciones eficientes para datos complejos

Contexto: 🙌

Una plataforma educativa necesita manejar un conjunto de datos donde cada alumno tiene nombre, edad, cursos y calificaciones. Debe organizarse esta información y procesarla para eliminar duplicados, calcular estadísticas y visualizar información estructurada de forma eficiente.

Consigna: ✍️

Codificar una rutina que utilice tuplas, sets, comprensiones y estructuras anidadas para almacenar y procesar información académica.

Tiempo 🕐: *30 Minutos*

Colecciones eficientes para datos complejos

Paso a paso: ⚙️

1. Crear un set con los nombres únicos de alumnos registrados (simular duplicados y eliminarlos).
2. Crear una estructura anidada (lista de diccionarios) con nombre, edad, cursos (lista) y calificaciones (diccionario).
3. Utilizar comprensión de listas para obtener una lista de alumnos mayores a 22 años.
4. Utilizar comprensión de diccionarios para filtrar solo calificaciones mayores a 90.
5. Usar una función con retorno múltiple (tupla) para devolver promedio y cantidad de cursos de un alumno.
6. Mostrar resultados con formato limpio.

¿Alguna consulta?

alkemy

Resumen

¿Qué logramos en esta clase?

- ✓ ***Aprendimos a usar tuplas como estructuras inmutables***
- ✓ ***Aplicamos sets para eliminar duplicados y realizar operaciones de conjunto***
- ✓ ***Usamos comprensiones para transformar datos con una sintaxis más clara***
- ✓ ***Creamos estructuras complejas combinando listas, diccionarios y otros tipos***
- ✓ ***Aplicamos buenas prácticas para elegir la estructura más eficiente***

¡Ponte a prueba!

Momento de ejercitación

Te invitamos a aprovechar esta última sección del espacio sincrónico para realizar de manera individual las ***actividades disponibles en la plataforma.*** *Estas propuestas son clave para afianzar lo trabajado y* ***forman parte obligatoria del recorrido de aprendizaje.***

💪 Análisis de caso ——— 💪 Selección Múltiple

💪 Comprensión lectora

Si al resolverlas surge alguna duda, compartela o tráela al próximo encuentro sincrónico.

alkemy

¡Muchas gracias!

alkemy