Programación 1

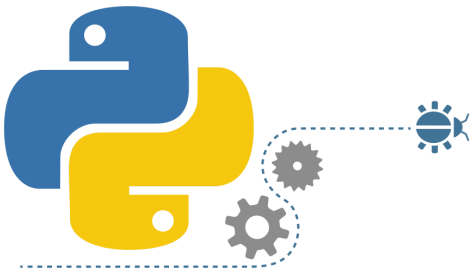
1º Año

Clase N°8 : Estructura de datos

Contenido

- ¿Qué son las estructuras de datos y para qué se usan?
- Vectores (arrays unidimensionales): cómo se crean y para qué sirven.
- Operaciones con listas: cómo asignar, leer, recorrer y actualizar datos.
- Arrays de más de una dimensión: tablas (bidimensionales) y estructuras más complejas (multidimensionales).
- Cómo se guardan los arrays en la memoria.
- Estructuras dinámicas en Python: Listas (list), Tuplas (tuple), Conjuntos (set) y Diccionarios (dict).

1. Presentación



¡Bienvenidos y bienvenidas a una nueva clase de Programación en Python!

Hoy nos adentramos en un tema esencial para el manejo eficiente de la información en nuestros programas:

Las estructuras de datos.

Primero vamos a responder una pregunta clave:

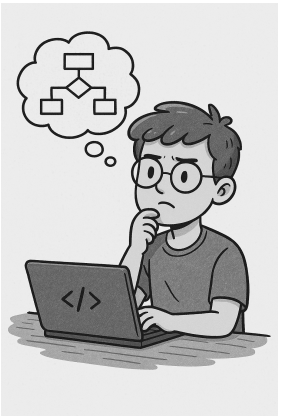
¿Qué son las estructuras de datos y para qué se usan?

Son formas de organizar los datos dentro de un programa para que podamos trabajar con ellos de forma más ordenada, rápida y clara.

Vamos a aprender cómo guardar, ordenar y usar datos en Python. Veremos desde listas simples hasta estructuras más completas como diccionarios y matrices. 📊

Al terminar la clase, vas a saber cómo organizar la información de forma clara y práctica, algo clave para resolver problemas reales y hacer programas más ordenados y fáciles de mejorar. 🚀

Pregunta para reflexionar



Imagina que estás armando una biblioteca en tu casa. Tenés cientos de libros desordenados por toda la sala: algunos en pilas, otros en cajas o tirados en la mesa.

📌 ¿Cómo te organizarías para encontrarlos fácilmente más adelante?

📌 ¿Los agruparías por género, por autor, por año o por tamaño?

📌 ¿Cómo los guardarías para acceder rápido y sin confusión?

(Tomémonos unos minutos para pensar)

Lo que acabas de imaginar es, en esencia, organización de datos.

Hoy vamos a aprender cómo aplicarla en programación usando estructuras como listas, tuplas, diccionarios y arrays.

🎥 Vídeo Tutorial

Te proponemos mirar el siguiente video introductorio sobre estructuras de datos en Python. Verás ejemplos visuales de cómo se usan vectores, listas, diccionarios y más.

🎥 **Introducción a las estructuras de datos**

📝 Te proponemos mirar el video y luego responder:

Preguntas para reflexionar después del video

➔ ¿Por qué creés que existen distintos tipos de estructuras de datos y no una sola para todo?

➔ ¿Qué ventajas tiene usar una lista en lugar de muchas variables sueltas?

➔ ¿Cómo influye la forma de almacenar los datos en la facilidad para acceder o modificarlos?

➔ ¿Podés pensar en algún ejemplo de tu vida diaria donde usás, sin darte cuenta, una estructura de datos?

(Ejemplo: una lista de compras, una agenda, tu bandeja de entrada, etc.)

2. Desarrollo

2.1 Arrays unidimensionales (Vectores)

Los **arrays unidimensionales**, conocidos como **vectores**, son estructuras que nos permiten guardar varios datos del mismo tipo en una sola variable. Los datos se guardan uno al lado del otro en la memoria y se accede a cada uno usando un **número de posición** (llamado índice).

Ejemplo en Python

```
mi_vector = [1, 2, 3, 4, 5]
```

Este código crea una lista con 5 números enteros. Cada número se puede acceder con su posición: `0` para el primero, `1` para el segundo, y así sucesivamente.

```
primer_elemento = mi_vector[0]  # Da 1
```

Aquí accedemos al primer valor del vector y lo guardamos en la variable `primer_elemento`.

```
mi_vector[2] = 10  # Ahora el vector es [1, 2, 10, 4, 5]
```

Este código cambia el tercer valor del vector (posición 2) a 10.

Operaciones comunes con vectores

Asignar un valor

```
mi_vector[2] = 7
```

Reemplaza el valor que está en la posición 2 por un nuevo valor (7).

Leer y escribir datos

```
valor = mi_vector[1]   # Lee el segundo elemento
mi_vector[3] = 10       # Cambia el cuarto valor a 10
```

El primer ejemplo guarda en `valor` lo que hay en la posición 1. El segundo cambia el valor en la posición 3 por 10.

Recorrer (iterar)

```
for elemento in mi_vector:
    print(elemento)
```

Este bucle va tomando cada valor del vector **`mi_vector`** uno por uno y lo imprime por pantalla.

Actualizar todos los valores

```
for i in range(len(mi_vector)):
    mi_vector[i] += 1
```

Aquí usamos **`range`** para recorrer todas las posiciones del vector, y a cada valor le sumamos 1.

Actividad práctica

Crear un vector con 4 notas de un alumno y mostrar el promedio por pantalla.

Reflexión

¿Por qué es útil tener todos los datos relacionados (como notas o edades) en una sola estructura? ¿Qué pasaría si usáramos una variable para cada uno?

2.2 Arrays de varias dimensiones

Un **array bidimensional** es como una tabla con filas y columnas. Se usa para representar datos organizados, como en una hoja de cálculo o una matriz.

Ejemplo en Python

```
matriz = [
    [1, 2, 3],
    [4, 5, 6]
]
```

Creamos una matriz con 2 filas y 3 columnas. Cada fila es una lista. Accedemos a sus valores usando dos índices:

```
print(matriz[1][2])  # Fila 2, columna 3
```

Este código muestra el valor 6, que se encuentra en la fila 2 y la tercera columna.

Recorrer una matriz

```
for fila in matriz:
    for valor in fila:
        print(valor)
```

Se recorre cada fila de la matriz, y dentro de esa fila, cada valor. Así se imprimen todos los números de la matriz.

Arrays Multidimensionales

Un array de más de dos dimensiones puede representar datos aún más complejos, como imágenes o estructuras 3D.

Ejemplo 3D:

```
cubo = [
    [[1, 2], [3, 4]],
    [[5, 6], [7, 8]]
]
print(cubo[1][0][1])  # Devuelve 6
```

Aquí tenemos una estructura tridimensional. El primer índice selecciona el "bloque" (en este caso, el segundo bloque), el segundo índice selecciona la fila, y el tercero el valor dentro de esa fila. Resultado: 6.

Actividad práctica

Representar una tabla de multiplicar del 1 al 3 usando una matriz y mostrarla.

Reflexión

¿En qué situaciones te parece útil usar una estructura como una matriz? ¿Qué tipo de datos reales podrían organizarse así?

2.3 Almacenamiento de arrays en memoria

Cuando usamos arrays (o vectores) en programación, no solo nos interesa cómo se ven desde el código, sino también cómo se

almacenan en la memoria de la computadora. Comprender esto nos ayuda a escribir programas más eficientes.

Los arrays guardan sus elementos en **espacios de memoria contiguos**, es decir, uno al lado del otro. Esto significa que si tenemos un array con 5 números, como:

mi_vector = [10, 20, 30, 40, 50]

Python (o el lenguaje que estemos usando) reserva 5 espacios de memoria uno detrás del otro para guardar esos valores. Gracias a esta organización:

- Podemos acceder rápidamente a cualquier elemento con su índice.
- Se mejora la velocidad de lectura y escritura.

¿Por qué importa esto?

- Si accedemos muchas veces a los elementos de un array, esta forma de almacenamiento contiguo hace que el acceso sea **muy rápido**.

- También permite recorrer los datos de forma más eficiente, algo clave en algoritmos de búsqueda o procesamiento masivo.

2.4 Colecciones en Python (estructuras de datos dinámicas)

En muchos programas no sabemos de antemano cuántos datos vamos a manejar. Por eso, Python nos ofrece colecciones dinámicas: estructuras que permiten agregar, quitar o modificar elementos mientras el programa está en marcha.

A diferencia de los arrays tradicionales, estas colecciones se ajustan automáticamente al tamaño necesario y son muy flexibles.

Principales tipos:

- Listas (`list`): ordenadas y modificables.
- Tuplas (`tuple`): ordenadas pero no se pueden modificar.
- Conjuntos (`set`): sin orden, sin elementos repetidos.
- Diccionarios (`dict`): organizan datos en pares clave-valor.

Estas estructuras usan memoria dinámica, por lo que son ideales para trabajar con información que cambia durante la ejecución del programa.

Listas (`list`)

Las listas en Python son colecciones ordenadas y mutables. Esto significa que los elementos mantienen el orden en que se agregan, y además se pueden modificar en cualquier momento: podemos agregar, quitar o cambiar elementos mientras el programa se ejecuta.

Se definen usando corchetes [] y los elementos se separan por comas.

Por ejemplo:

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

frutas.append("pera")

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

📌 **Explicación del código**:

- Se crea una lista llamada `frutas` con dos elementos iniciales: `"manzana"` y `"banana"`.
- Usamos el método `.append()` para **agregar** `"pera"` al final de la lista.
- Al imprimir la lista con `print()`, vemos el contenido actualizado con las tres frutas.

✅ Las listas son muy útiles cuando necesitamos manejar conjuntos de datos que pueden **crecer o cambiar** a lo largo del programa, como una lista de tareas, productos o nombres de usuarios.

Actividad práctica

Crear una lista con tus 3 comidas favoritas, agregar una más y mostrar el resultado.

Tuplas (`tuple`)

Las **tuplas** son colecciones **ordenadas e inmutables**. Esto significa que los elementos mantienen un orden fijo, pero **no se pueden modificar una vez creada la tupla**: no se pueden agregar, quitar ni cambiar valores.

Se definen usando paréntesis `()` y los elementos se separan por comas.

Por ejemplo:

```
tupla_colores = ("rojo", "verde", "azul")

print(tupla_colores[1])  # verde
```

📌 **Explicación del código**:

- Se crea una tupla llamada `tupla_colores` con tres elementos: `"rojo"`, `"verde"` y `"azul"`.
- Usamos el índice `[1]` para acceder al **segundo** color (recordá que los índices empiezan desde 0).
- Al ejecutar el `print()`, se muestra `"verde"`.

✅ Las tuplas son útiles cuando queremos **proteger los datos** y asegurarnos de que no cambien, por ejemplo: coordenadas geográficas, configuraciones que no deben alterarse, o claves que sirven como identificadores.

Actividad práctica

Crear una tupla con tres nombres de ciudades y mostrar la segunda.

Conjuntos (`set`)

Los **conjuntos** en Python son colecciones **no ordenadas** y que **no permiten elementos repetidos**. Son ideales cuando queremos almacenar datos únicos y no nos importa el orden en que aparecen.

Se pueden crear con llaves `{}` o usando la función `set()`.

Por ejemplo:

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

print(numeros)  # {1, 2, 3, 4}
```

📌 **Explicación del código**:

- Creamos un conjunto llamado `numeros` con los valores 1, 2, 3 (repetido) y 4.
- Aunque escribimos dos veces el número 3, el conjunto **automáticamente elimina los duplicados**.
- Al imprimirlo, vemos solo los valores únicos: `{1, 2, 3, 4}`.
- El orden puede variar cada vez que lo ejecutes, ya que los conjuntos **no garantizan un orden fijo**.

✅ Los conjuntos son muy útiles cuando queremos eliminar duplicados de una lista, comparar elementos comunes entre colecciones (intersección), o verificar si un valor existe de forma rápida y eficiente.

Actividad práctica

Crear un set con números repetidos y mostrar cómo los elimina automáticamente.

Diccionarios (`dict`)

Los **diccionarios** son estructuras de datos que almacenan **pares de clave y valor**. Esto significa que a cada valor se lo identifica con una clave única. Aunque no mantienen un orden específico, permiten acceder rápidamente a la información usando esa clave.

Se definen usando llaves `{}`, y cada par se escribe con la forma `clave: valor`.

Por ejemplo:

```
datos = {"nombre": "Ana", "edad": 30}
print(datos["nombre"])  # Ana
```

📌 **Explicación del código**:

- Creamos un diccionario llamado `datos` con dos claves: `"nombre"` y `"edad"`.
- Cada clave tiene un valor asociado: `"Ana"` para `"nombre"` y `30` para `"edad"`.
- Al escribir `datos["nombre"]`, accedemos directamente al valor asociado a la clave `"nombre"`, que es `"Ana"`.

✅ Los diccionarios son especialmente útiles cuando necesitamos **relacionar información**, como nombres con edades, productos con precios, usuarios con contraseñas, etc. Nos permiten buscar, agregar, actualizar o eliminar datos de forma rápida utilizando las claves.

Actividad práctica

Crear un diccionario con tu nombre, edad y ciudad. Mostrar el valor de la ciudad.

Actividad de cierre

Vamos a realizar una actividad para reforzar cómo se utilizan las estructuras de datos en Python (listas, diccionarios, tuplas, etc.) y cómo nos ayudan a organizar la información de forma clara y eficiente.

PASO 1 >> Usamos la IA para generar un ejemplo

Abrí Chat GPT y escribí:

🗨️ ***"Creá un diccionario en Python que almacene nombres de personas como claves y sus edades como valores. Luego, agregá una persona nueva al diccionario y mostrá todas las claves y valores con un bucle."***

Copiá el código que te da y léelo con atención.

PASO 2 >> Analizamos el código

Explicá con tus palabras **qué hace el código, línea por línea**.

- ¿Qué estructura de datos se usó?
- ¿Cómo se acceden y modifican los datos?
- ¿Para qué sirve el bucle que recorre el diccionario?

Luego, respondé:

🧠 ¿Qué ventajas tiene usar un diccionario en este caso, en lugar de dos listas separadas?

📌 ¿Cómo podrías hacer algo parecido con listas o tuplas?

👉 **Importante**: Esta parte hacela vos, sin usar Chat GPT. Queremos ver cómo lo entendiste y cómo aplicas los conceptos vistos hoy (definición, parámetros, return, etc.).

PASO 3 >> Compartimos en el foro

Subí al foro de intercambio de la Clase N°8 lo siguiente:

1. ✅ El código que te generó ChatGPT.
2. ✍️ Tu explicación línea por línea.
3. 🧠 Una comparación con otra estructura de datos (si se te ocurre una forma distinta de hacerlo).

Instancia de Autoevaluación

Cuestionario de Moodle

Para finalizar esta clase, te invitamos a realizar un cuestionario de autoevaluación que te permitirá fijar los conocimientos aprendidos y reflexionar sobre los temas vistos.

🧠 Este cuestionario te ayudará a:

- Reforzar los contenidos claves de la clase.
- Identificar los conceptos que ya dominas.
- Detectar posibles dudas para repasar.

📝 Criterios de evaluación

➔ El cuestionario te brindará el resultado final.
➔ La nota para aprobar es un 6 (seis) sobre 10.
➔ Tendrás 10 preguntas multiple choice.
➔ Cantidad de intentos: 2 intentos permitidos.
➔ Es importante que leas esto: Si desaprobaste en el primer intento y aprobaste el segundo, la segunda nota se toma como referencia. Ejemplo: Si sacaste un 5 en el primer intento y 7 en el segundo, tu nota quedará en 7.
➔ Te recomendamos prestar mucha atención, ya que algunas preguntas incluyen análisis de código en Python.
➔ Tomate el tiempo necesario para leer bien cada enunciado. Si lo necesitás, podés probar el código antes en tu entorno de Visual

Studio Code para asegurarte de comprenderlo y responder con seguridad.

👉 **Importante**: la realización del cuestionario es obligatoria y forma parte del proceso de aprendizaje.

📝 ¡Accedé al cuestionario desde el enlace correspondiente y completalo antes de la fecha límite indicada!

Link cuestionario - Comisión A

Link cuestionario - Comisión C

Link cuestionario - Comisión B

Link cuestionario - Comisión D

3. Cierre

En esta clase nos sumergimos en uno de los aspectos más fundamentales de la programación: las estructuras de datos. Aprendimos cómo organizar y almacenar información de manera eficiente utilizando vectores, matrices, listas, tuplas, conjuntos y diccionarios. Exploramos cómo acceder, modificar y recorrer estos datos según sus características, reconociendo cuándo conviene usar cada tipo.

Estos conceptos no sólo son esenciales para representar datos dentro de un programa, sino que también forman la base para diseñar soluciones más claras, ordenadas y eficientes. Saber elegir la

estructura adecuada nos permite reducir la complejidad del código, aprovechar mejor la memoria y facilitar el procesamiento de información.

Comprender cómo funcionan estas estructuras es un paso clave para desarrollar programas más potentes, escalables y preparados para manejar problemas reales del mundo del software.

4. Bibliografía

- Joyanes Aguilar, L. (2008). *Fundamentos de programación: Algoritmos, estructura de datos y objetos* (4.ª ed.). McGraw-Hill.
- Bonanata, M. (2003). *Programación y algoritmos: Aprenda a programar en C y Pascal*. M.P. Ediciones. *(Manuales Users)*
- Cantone, D. (2003). *La biblia del programador: Implementación y debugging*. M.P. Ediciones. *(Manuales Users .code)*
- OpenAI. (2025). ChatGPT (versión GPT-4). Recuperado de https://chat.openai.com