Programación Avanzada
IIC2233 2024-1

Hernán Valdivieso - Daniela Concha - Francisca Ibarra - Dante Pinto - Francisca Cattan

Anuncios

1. ¡Hagan *push* de su tarea!
2. Hoy tenemos la primera experiencia.
3. No olviden revisar las *issues* importantes.
4. Revisen el compilado antes de abrir una nueva *issue*.
5. ¡¡Hagan *push* de su tarea!!

Repaso

Estructuras de datos

- Forma especializada de agrupar datos.
- Almacenamiento, acceso y utilización eficiente.
- ¿Qué estructura es mejor para cada caso?

Lo básico

Estructuras secuenciales

- Orden secuencial de elementos.
- Garantizan un recorrido ordenado y eficiente de los elementos.
- Algunas de estas estructuras son:
    - Tuplas
    - *Named tuples*
    - Listas (Arreglos)
    - *Stacks*
    - Colas

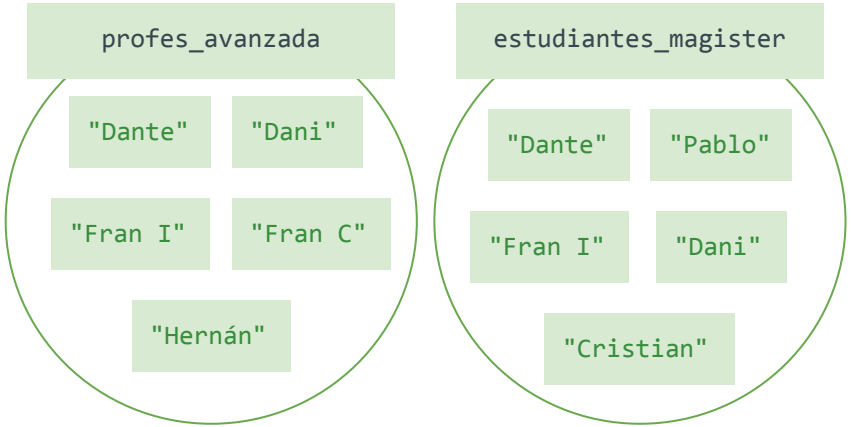
Estructuras no secuenciales

- No hay orden entre elementos.
- Búsqueda de elementos muy eficiente.
- Algunas de estas estructuras son:
    - *Sets* (Conjuntos)
    - Diccionarios
    - *Defaultdict*

Sets (Conjuntos)

- No hay orden entre elementos.
- Búsqueda de elemento específico muy eficiente.
- No permite duplicados.
- Solo permite elementos inmutables.

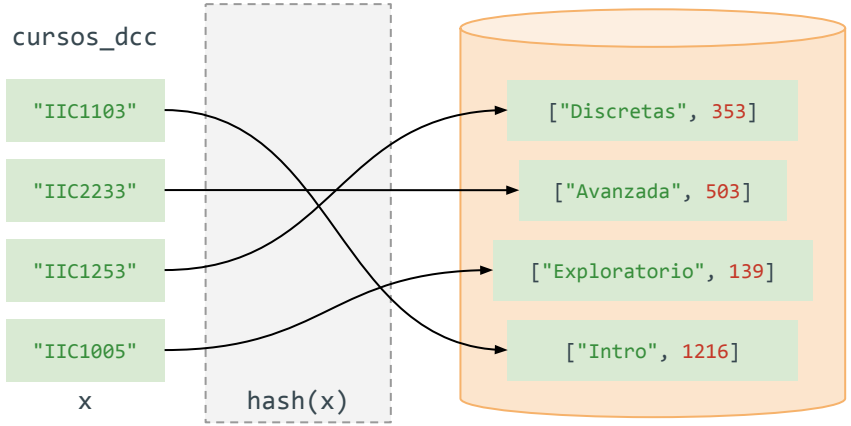
```
estudiantes_magister = {"Cristian", "Dante",
                        "Fran I", "Pablo", "Dani"}

profes_avanzada = set(profesores)

print("Cristian" in profes_avanzada)
print(profes_avanzada & estudiantes_magister)
```



Diccionarios

- Almacena pares: llave-valor.
- Búsqueda de llave específica muy eficiente.
- Llaves no pueden estar duplicadas.
- Solo permite elementos inmutables como llaves.
- Valor puede ser cualquier elemento.

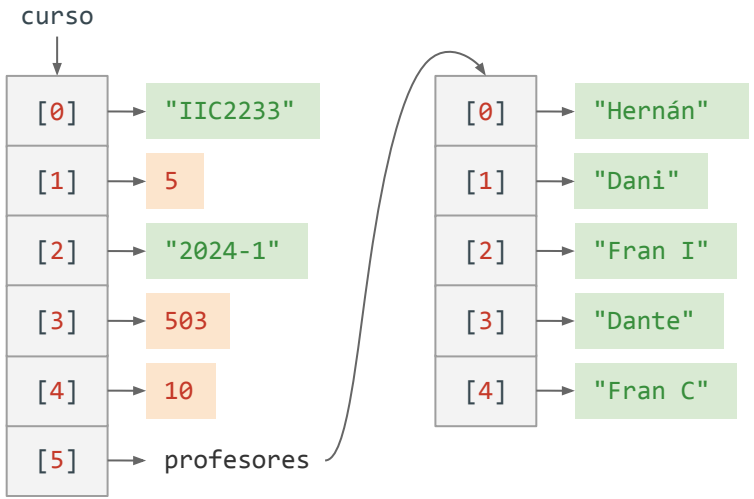
```
cursos_dcc = { "IIC1103": ["Intro", 1216],
               "IIC2233": ["Avanzada", 528],
               "IIC1253": ["Discretas", 353],
               "IIC1005": ["Exploratorio", 139]}

print(cursos_dcc["IIC2233"])
print(cursos_dcc["IIC1103"][1])
```



Tuplas

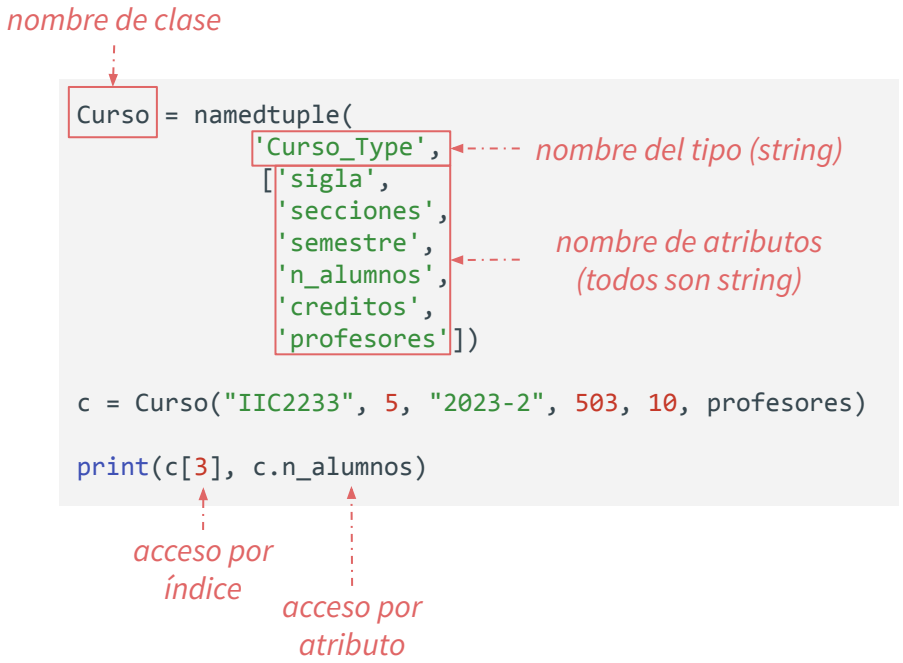
- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Immutable.
- Suelen utilizarse para agrupar elementos. heterogéneos.

```
curso = ("IIC2233", 5, "2023-2", 503, 10, profesores)
print(curso[3])
```



Named tuples

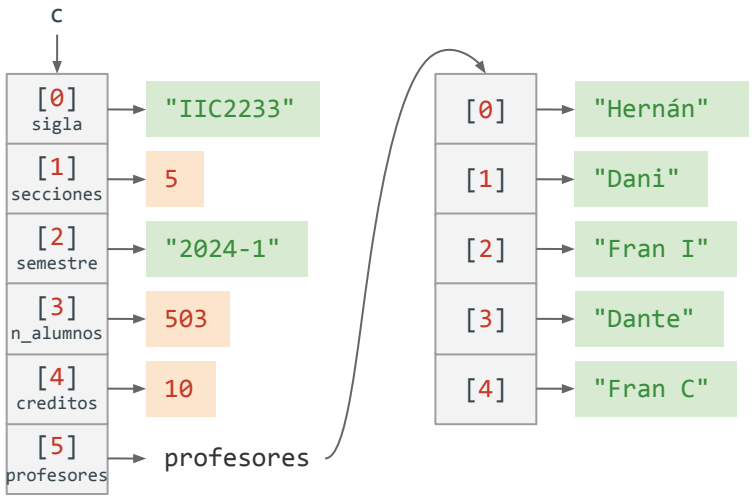
- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Immutable.
- **Cada posición tiene un nombre (atributo).**



Named tuples

- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Immutable.
- **Cada posición tiene un nombre (atributo).**

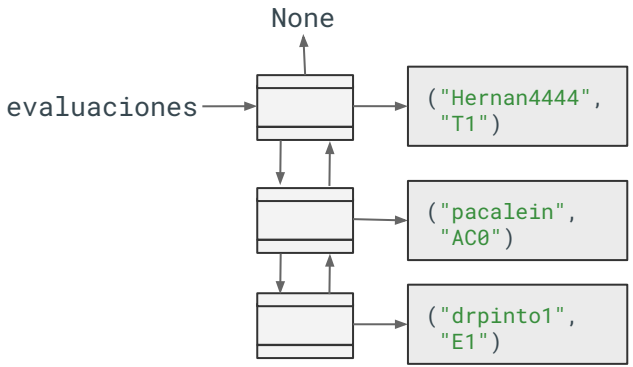
```
c = Curso("IIC2233", 5, "2023-2", 503, 10, profesores)
print(c[3], c.n_alumnos)
```



Colas

- Orden secuencial de elementos.
- Inserción/eliminación eficiente en el extremo de la cola.
- Mutable
- Las *deque* de Python son eficientes en ambos extremos.

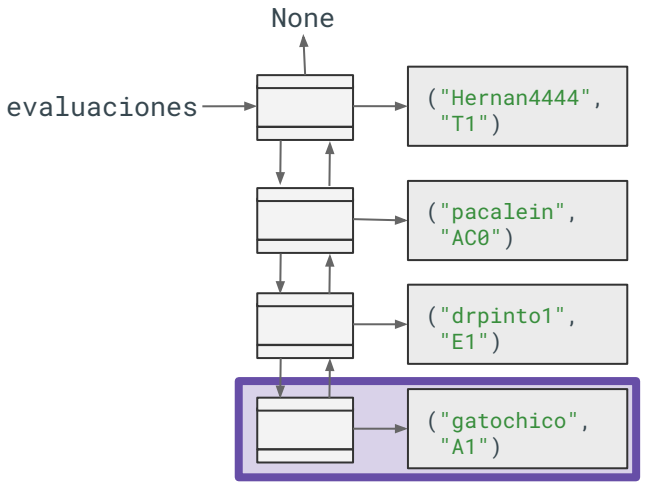
```
from collections import deque

evaluaciones = deque([("Hernan4444", "T1"),
                      ("pacalein", "AC0"),
                      ("drpinto1", "E1")])
```



Colas

- Orden secuencial de elementos.
- Inserción/eliminación eficiente en el extremo de la cola.
- Mutable
- Las *deque* de Python son eficientes en ambos extremos.

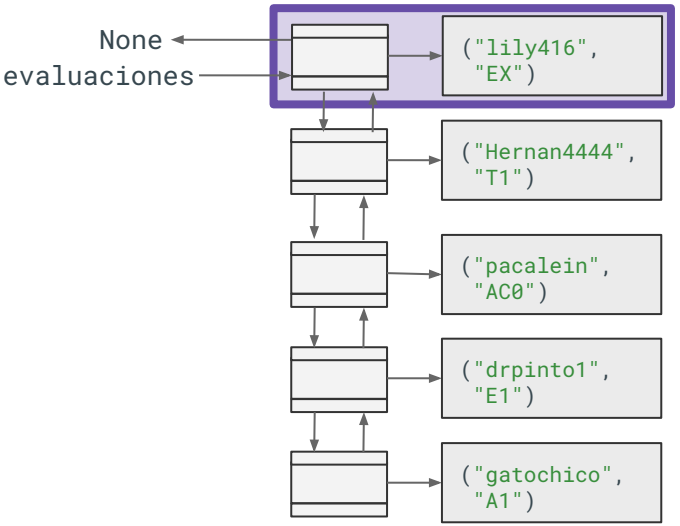
```
from collections import deque

evaluaciones = deque([("Hernan4444", "T1"),
                      ("pacalein", "AC0"),
                      ("drpinto1", "E1")])
evaluaciones.append(("gatochico", "A1"))
```



Colas

- Orden secuencial de elementos.
- Inserción/eliminación eficiente en el extremo de la cola.
- Mutable
- Las *deque* de Python son eficientes en ambos extremos.

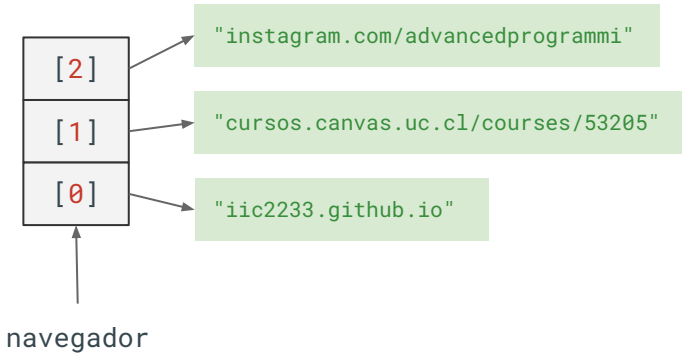
```
from collections import deque

evaluaciones = deque([("Hernan4444", "T1"),
                      ("pacalein", "AC0"),
                      ("drpinto1", "E1")])
evaluaciones.append(("gatochico", "A1"))
evaluaciones.appendleft(("lily416", "EX"))
```



Stack

- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Mutable.
- Inserción y eliminación al final.

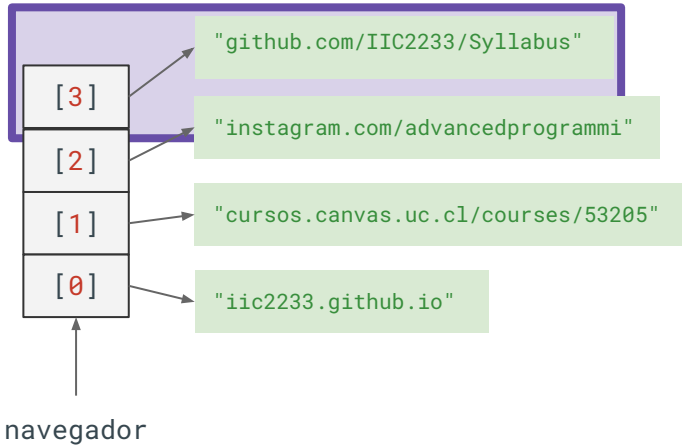
```
navegador = [
    "iic2233.github.io",
    "cursos.canvas.uc.cl/courses/53205",
    "instagram.com/advancedprogrammi"
]
```



Stack

- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Mutable.
- Inserción y eliminación al final.

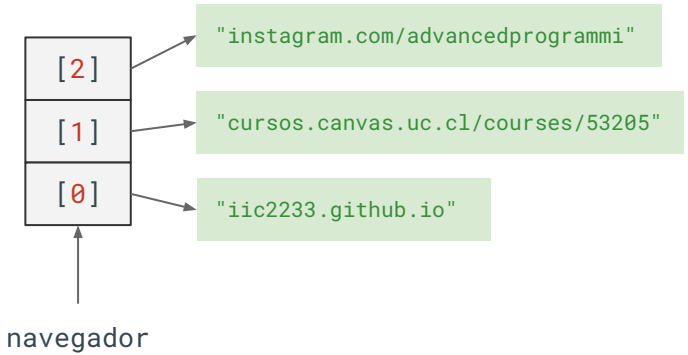
```
navegador = [
    "iic2233.github.io",
    "cursos.canvas.uc.cl/courses/53205",
    "instagram.com/advancedprogrammi"
]
navegador.append("github.com/IIC2233/Syllabus")
```



Stack

- Orden secuencial de elementos.
- Búsqueda de i-ésimo elemento muy eficiente.
- Mutable.
- Inserción y eliminación al final.

```
navegador = [
    "iic2233.github.io",
    "cursos.canvas.uc.cl/courses/53205",
    "instagram.com/advancedprogrammi"
]
navegador.append("github.com/IIC2233/Syllabus")
navegador.pop() # Elimina el último
```



Estructuras secuenciales

Estructura	Mutable	*Hasheable*	
Lista	✔	✘	Permiten agregar, eliminar, modificar elementos.
Tupla	✘	✔*	Útiles para retornar múltiples valores y contener valores que no cambian.
Named Tuple	✘	✔*	Tuplas donde se puede acceder a cada posición mediante un nombre.
Colas	✔	✘	Eficientes para insertar o retirar desde sus extremos
Stacks	✔	✘	Inserción y eliminación son siempre en el tope del *stack*

Resumen

Estructura	Insertar	Búsqueda por índice	Búsqueda por llave	Búsqueda por valor
Lista	✔	✔✔✔	✘	✔
Tupla	✘	✔✔✔	✘	✔
Diccionario	✔✔✔	✘	✔✔✔	✔
Set (Conjunto)	✔✔✔	✘	✔✔✔	✘

✔✔✔ Se puede y es eficiente.

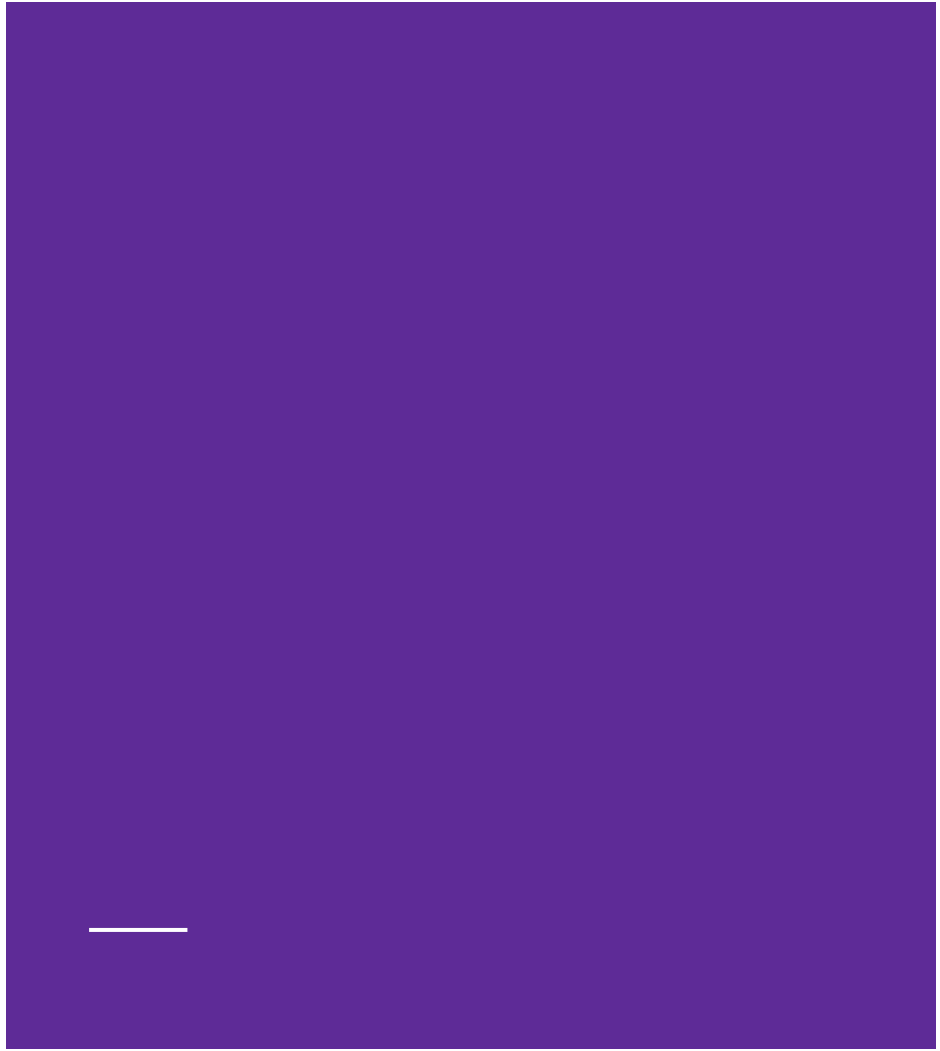
✔ Se puede pero no siempre es eficiente.

✘ No se puede.

Experiencia

Experiencia 1

Estructuras de datos *built-in*



Experiencia 1: ¿Qué vamos a hacer?

1. Corregiremos los errores de una solución parcial a un problema dado.
2. Verificaremos nuestra solución a través de un conjunto de tests.
3. Aplicaremos los contenidos de estructuras de datos para encontrar una solución eficiente a nuestro problema.
4. Completaremos las funcionalidades ausentes en la solución parcial.

DCCancionero

Al enterarse del éxito que han tenido las diferentes plataformas de *streaming* de música, el DCC decidió crear un nuevo competidor, el DCCancionero.

Lamentablemente, por temas de tiempo, su implementación está incompleta y llena de errores, por lo que necesitan nuestra ayuda para completarla.

¿En qué consiste el DCCancionero?

Es una aplicación pensada para el *streaming* de música y debe ser capaz de:

- Guardar las canciones, *playlist* e historial de reproducciones del usuario.
- Cargar información de canciones desde una base de datos.
- Manejar una cola de reproducción, permitiendo reproducir, eliminar y agregar canciones de la cola.
- Administrar *playlists* del usuario, incluyendo: creación de nuevas *playlist,* reproducción de *playlists,* y edición de las *playlists* existentes.

Manejo de datos

El manejo de los datos del usuario de DCCancionero ya se encuentra implementado, incluyendo la carga de datos.

Puedes comprobar su correcto funcionamiento agregando líneas de código en `main.py` y/o ejecutando el archivo de `tests.py`.

La lógica detrás de esta funcionalidad, se encuentra implementada en el `init` de la clase `DCCancionero` y en su método `cargar_canciones`, que se encarga de leer la base de datos en `canciones.dat` y de guardar una lista de estas canciones.

Manejo de datos

El manejo de los datos del usuario de DCCancionero ya se encuentra implementado, incluyendo la carga de datos.

Puedes comprob digo en `main.py` y/o ejecutando e

Vamos a ver el código que tenemos 👨‍💻👨‍💻

La lógica detrás c el `init` de la clase `DCCancionero` y en su método `cargar_canciones`, que se encarga de leer la base de datos en `canciones.dat` y de guardar una lista de estas canciones.

Cola de reproducción

Deberás corregir los siguientes métodos:

- `agregar_a_cola(self, *args):`
  - En caso de no recibir argumentos, agrega a la cola la última canción del historial del usuario.
  - En caso de recibir exactamente un argumento, este debe agregarse a la cola.
  - En caso de recibir dos o más argumentos, el método no hace nada.
- `escuchar_siguiente_cancion(self):`
  - El método debe obtener la siguiente canción de la cola e imprimir un mensaje avisando al usuario que comienza la reproducción de la canción.
  - Si la cola está vacía, se debe imprimir un mensaje de error informando la situación.

Cola de reproducción

Deberás corregir los siguientes métodos:

- `agregar_a_cola(self, *args):`
  - En caso el historial del usuario.
  - En caso la cola.
  - En caso de recibir dos o más argumentos, el método no hace nada.
- `escuchar_siguiente_cancion(self):`
  - El método debe obtener la siguiente canción de la cola e imprimí un mensaje avisando al usuario que comienza la reproducción de la canción.
  - Si la cola está vacía, se debe imprimir un mensaje de error informando la situación.

Vamos a corregir el código que tenemos 👩‍💻👨‍💻

Manejo de *playlists*

Deberás corregir el siguiente método:

- `crear_playlist(self, nombre, canciones=None):`
  - Se debe crear una nueva *playlist*, inicialmente vacía, usando una estructura de datos apropiada, donde cada canción puede aparecer solamente una vez.
  - La *playlist* creada debe guardarse dentro de la información del usuario, permitiendo acceder a ella a partir de su nombre.
  - En caso de recibir el argumento canciones, se deben agregar las canciones de esta lista a la *playlist*.
  - Finalmente, se debe imprimir un mensaje informando la correcta creación de la *playlist*.

Manejo de *playlists*

Deberás completar los siguientes métodos:

- `escuchar_playlist(self, nombre):`
  - Si la *playlist* existe, se deben agregar todas sus canciones a la cola de reproducción y se debe imprimir un mensaje informando de la operación.
  - En caso de que no exista la *playlist*, se debe imprimir un mensaje informando al usuario.

- `agregar_a_playlist(self, nombre_playlist, cancion):`
  - Se debe agregar la canción “cancion” a la playlist “nombre_playlist”
  - Si la playlist no existe, se debe imprimir un mensaje de error.

Manejo de *playlists*

Deberás completar los siguientes métodos:

- `escuchar_playlist(self, nombre):`
  - Si la *play* reproducción y se debe im
  - En caso rmando al usuario.

Vamos a completar el código que tenemos 🤦‍♂️🤦‍♂️

- `agregar_a_playlist(self, nombre_playlist, cancion):`
  - Se debe agregar la canción “cancion” a la playlist “nombre_playlist”
  - Si la playlist no existe, se debe imprimir un mensaje de error.

Desafíos

¡Felicidades! Lograste reparar y completar todas las funcionalidades requeridas del DCCancionero. Si quieren continuar programando, te dejamos a estos desafíos.

Desafío 1: Almacenamiento de canciones.

Si te fijas en el código, en ningún momento se está haciendo uso de las canciones cargadas de la base de datos 🤯.

Lo anterior se debe a que están guardadas en una lista, lo que dificulta su acceso.

Utiliza una estructura de datos alternativa para almacenar canciones y modifica los métodos que reciben una canción para que aprovechen esta nueva estructura.

Desafío 2: Edición de la cola.

1. Actualmente si cometes un error o cambias de opinión sobre una canción la cola te obliga a escucharla antes de pasar a la siguiente. Implementa la eliminación de canciones en la cola.
2. Para implementar la cola estamos utilizando un *deque*. Aprovecha las características de esta estructura para incluir el “agregar a continuación” entre las funcionalidades de la cola.

Programación Avanzada
IIC2233 2024-1

Hernán Valdivieso - Daniela Concha - - Dante Pinto - Francisca Cattan