PONTIFICIA UNIVERSIDAD CATÓLICA DE CHILE
ESCUELA DE INGENIERÍA
DEPARTAMENTO DE CIENCIA DE LA COMPUTACIÓN

IIC2233 Programación Avanzada (I/2018)

Tarea 3

Entrega

- `Entregable Obligatorio`
  - **Fecha/hora:** sábado 28 de abril del 2018, 23:59 horas.
  - **Lugar:** Google Form `https://goo.gl/forms/vMfK2agtjQaz94Cx1`
- `Tarea Obligatorio`
  - **Fecha/hora:** domingo 6 de mayo del 2018, 23:59 horas.
  - **Lugar:** GitHub – Carpeta: `Tareas/T03/`
- `README.md Obligatorio`
  - **Fecha/hora:** lunes 7 de mayo del 2018, 23:59 horas.
  - **Lugar:** GitHub – `Carpeta: Tareas/T03/`

Objetivos

- Entender y aplicar el paradigma de programación funcional para resolver un problema.
- Manejar datos de forma eficiente usando herramientas de programación funcional.
- Aplicar el conocimiento de excepciones y su manejo para crear programas robustos.
- Escribir *tests* unitarios para verificar de forma automática el funcionamiento correcto de un programa.
- Aplicar la filosofía *"Do one thing and do it well"* en las funciones de la librería.
- Reforzar el conocimiento de ignorar archivos en Git mediante el uso del archivo `.gitignore`.
- Continuar en el proceso de familiarización con el paquete de interfaz gráfica `PyQt5`.

Índice

1. Introducción

La gran cadena de cines de la comuna de La Cruz, *"Cines Ruz"* está *ad portas* de la quiebra. Los motivos apuntan a malversación de fondos de parte del CEO, Cristian Ruz, para comprar una nueva oficina. Las noticias volaron y llegaron a oídos de su ayudante favorito, Hugo[1], quien recién había sido contratado en la dirección de informática del cine. Para no perder su nuevo trabajo, Hugo tuvo una gran idea: copiar diversos archivos con la bases de datos de películas, que incluye los actores de la película, el dinero recaudado y el género, entre otros datos más, para formar su propia cadena. ¡Ayuda a Hugo a trabajar con las bases y ver qué películas llevar a su nuevo cine!

2. *DCC Movie Database*

DCC Movie Database se compone de **tres** partes. Una librería, una interfaz de usuario y un módulo de *testing*:

- **Librería**

  En la primera parte, tendrás que implementar una librería la cual será posteriormente utilizada para responder una serie de consultas. Esta librería deberá cumplir con el paradigma de programación funcional y, además, deberá levantar excepciones cuando sea necesario.

- **Interfaz**

  La segunda parte consiste en utilizar una interfaz otorgada por los ayudantes, en donde se ingresarán diversas consultas a la base de datos que deberás responder utilizando la librería previamente creada. En este punto, deberás hacer uso de `try/except` para poder *atrapar* las diversas excepciones que genere la librería. Previo a empezar a recibir las consultas, será necesario realizar un preprocesamiento de la base de datos.

- ***Testing***

  De forma paralela, deberás crear un módulo de *testing* encargado de *testear* algunas consultas y excepciones de la librería.

2.1. Lectura del archivo

La base de datos que tu librería consta de **cuatro** archivos con formato CSV, los que contienen información de películas, junto con los géneros, actores y comentarios asociados a estas (`reviews.csv`, `movies.csv`, `genres.csv`, `actors.csv`).

Junto al enunciado se subieron **dos carpetas**, una llamada `Database` con aproximadamente doscientas películas y otra llamada `MiniDatabase` con 18 películas. Cada una de ellas dispone de esos cuatro archivos que se describirán a continuación.

2.1.1. Películas

La información acerca de las **películas** se encuentra en el archivo `movies.csv`, y cada fila representa a una película junto con su información. Cada columna del archivo representa lo indicado en la siguiente tabla:

[1] ¿Y quién lo conoce?

Nombre	Tipo de dato	Explicación
`id`	`integer`	Identificador único de cada película.
`title`	`string`	Título de la película.
`rating_imdb`	`string`	*Rating* proporcionado por la página IMDb.
`rating_metacritic`	`string`	*Rating* proporcionado por la página *Metacritic*.
`rating_rt`	`string`	*Rating* proporcionado por la página *Rotten Tomatoes*.
`box_office`	`string`	Dinero recaudado total, medido en dólares.
`date`	`integer`	Año de estreno de la película.

2.1.2. Géneros

La información acerca de los **géneros** de las películas se encuentra en el archivo `genres.csv`, y cada fila representa a una película y su género. Cada película puede tener uno o más géneros. Los géneros se pueden repetir en diferentes películas. Las columnas del archivo representan lo indicado en la siguiente tabla:

Nombre	Tipo de dato	Explicación
`id`	`string`	Identificador de cada película.
`genre`	`string`	Género asociado a esa película.

2.1.3. Actores

La información acerca de los **actores** que aparecen en cada película se encuentra en el archivo, y cada fila representa a una película y un actor o actriz que aparece en ella. Cada película puede tener uno o más actores y se pueden repetir un mismo actor en diferentes películas. Las columnas del archivo representan lo indicado en la siguiente tabla:

Nombre	Tipo de dato	Explicación
`id`	`string`	Identificador de cada película.
`actor`	`string`	Actor que aparece en esa película.

2.1.4. Comentarios

La información acerca de los **comentarios** que los usuarios hacen acerca de cada película se encuentra en el archivo `reviews.csv`, y cada fila representa a una película y un comentario acerca de ella. Cada película puede tener uno o más comentarios. Las columnas del archivo representan lo indicado en la siguiente tabla:

Nombre	Tipo de dato	Explicación
`id`	`string`	Identificador de cada película.
`review`	`string`	Comentario acerca de esa película.

Los comentarios presentes en esta tabla se encuentran alterados, y parte de tu trabajo consistirá en recuperar esa información (explicado en el punto 2.2.1)

2.2. Comentarios

2.2.1. Preprocesamiento

Tienes que determinar la popularidad de las películas basándote en los comentarios de los usuarios de la base de datos. Como no puedes leer uno por uno cada comentario, debes transformar los comentarios en un número que te indique la popularidad (o impopularidad) de una película.

Uno de tus queridos ayudantes, al obtener los comentarios desde las bases de datos, olvidó quitar algunas etiquetas HTML de estos. Una etiqueta HTML es el nombre de un elemento que está encerrado en corchetes angulares, (*i.e.* < y >), como por ejemplo , <a>, . Además pueden contener propiedades dentro de la etiqueta como, por ejemplo, <a href="www.google.cl">. Antes de procesar los comentarios debes borrar (o ignorar) todas las etiquetas HTML que quedaron en los comentarios, tengan o no propiedades. Es decir, se debe borrar todo (incluido) lo que se encuentra dentro de los corchetes angulares. Por ejemplo, si el comentario en `reviews.csv` es : *"This review may contain spoilers. <a href="/blott/film/little-miss-sunshine/" class="reveal js-reveal">I can handle the truth. "*. Luego de quitarle las etiquetas quedará así: *"This review may contain spoilers. I can handle the truth"*.

Sumado a esto, un *bot* infiltró comentarios en la base de datos, los cuales debes eliminar antes de determinar la popularidad de las películas. Hernán ha logrado interceptar un archivo llamado `vocabulary.txt` donde cada línea es una palabra del vocabulario del *bot*, de tal manera que puedes identificar cuales comentarios ha generado. Un mensaje del *bot* tendrá las siguientes características:

- Tiene 4 o más palabras de su vocabulario, sin contar repeticiones. Por lo tanto, si el vocabulario contiene las palabras *hola* y *chao*, y el comentario dice *hola hola hola*, sólo contamos **una** aparición.
- Repite 3 veces al menos una de las palabras de su vocabulario.
- Tiene entre 6 y 84 palabras totales.

Luego de eliminar los mensajes del *bot*, tendrás sólo comentarios de personas verdaderas por lo que puedes analizarlos para determinar si es un comentario positivo, neutro o negativo. Para determinarlo debes realizar los pasos mencionados en la sección 2.2.2.

2.2.2. Procesamiento

1. En el archivo `words.csv` encontrarás aquellas palabras clasificadas como positivas o negativas. Este archivo tiene dos columnas: `word` que contiene una palabra y `type` que indica si la palabra es `positive` o `negative`.
2. Determinar el porcentaje de palabras positivas y negativas del comentario como:

$$\begin{aligned} \%\text{ Palabras positivas} &= \frac{\text{Cantidad palabras positivas}}{\text{Total palabras comentario}} \cdot 100 \\ \%\text{ Palabras negativas} &= \frac{\text{Cantidad palabras negativas}}{\text{Total palabras comentario}} \cdot 100 \end{aligned}$$

3. Un comentario se considerará positivo si cumple **alguna** de las siguientes condiciones:
   a) % Palabras positivas ≥ 60
   b) % Palabras positivas ≥ 40 y, al mismo tiempo, % Palabras negativas ≤ 20
4. Un comentario se considerará negativo si cumple **alguna** de las siguientes condiciones:
   a) % Palabras negativas ≥ 60
   b) % Palabras negativas ≥ 40 y, al mismo tiempo, % Palabras positivas ≤ 20
5. En otro caso, se considerará como un comentario neutral.

2.3. Consultas

Uno de los servicios que ofrece la nueva cadena creada por Hugo es la búsqueda eficiente de películas bajo ciertos criterios. Es por ello que *DCC Movie Database* deberá disponer de las funciones que sean necesarias para responder a una serie de consultas.

Para empezar, en la base de datos existen varios tipos de *rating*, los cuales serán mostrados a continuación con sus respectivos parámetros y formato en la base de datos. Al momento de realizar las consultas, estos datos deben ser convertidos a una base de 100 **sólo cuando la consulta *requiera* utilizar alguno de los *ratings***. Si se realiza una consulta que no utiliza los *ratings* —por ejemplo, sólo ocupa `date`— entonces no debes transformar los *ratings*, puesto que eso sería procesamiento innecesario.

Fuente de rating	Nombre alternativo	Calificación del rating	Ejemplo	Equivalencia
Internet Movie Database	IMDb	K/10	6.5/10	65
Rotten Tomatoes	RT	K %	49 %	49
Metacritic	MC	K/100	51/100	51

En las consultas, se debe poder indicar el *rating* como el nombre de **fuente de *rating*** o usando el **nombre alternativo**. Por ejemplo, el resultado al buscar con `"Metacritic"` debe ser el mismo que al buscar con `"MC"`. Para esta tarea, el nombre de *rating* indicado tiene que ser exactamente igual a la **fuente de *rating*** o **nombre alternativo**. Si hubiese alguna letra diferente como mayúscula en vez de minúscula o algún espacio adicional, se considera como un parámetro inválido.

El promedio de los diferentes `rating_type` se obtiene mediante la conversión de cada tipo de *rating* a la equivalencia, y el promedio es la suma de los tres tipos dividido en tres. En el ejemplo, el *rating* promedio sería:

$$\textit{Rating}\ \text{promedio} = \frac{65 + 49 + 51}{3} = 55$$

2.3.1. Formato de consulta

El formato de las consultas será una lista cuyo primer elemento es el nombre de la consulta y los siguientes k elementos corresponden a los argumentos que requiere la consulta. Es decir:

```
["nombre", arg_1, arg_2, ... , arg_k]
```

El programa a realizar debe soportar el ingreso de varias consultas. Por lo tanto, el programa recibirá una lista de consultas en la forma:

```
[[consulta_1], [consulta_2], ... , [consulta_k]]
```

2.3.2. Consultas que retornan otra base de datos

Estas consultas retornan películas de la misma forma de la base de datos, por lo que debe ser posible tomar la salida de estas consultas y dárselas a otras consultas posteriores para continuar el procesamiento. Para describir cada tipo de dato de las consulta, usaremos el formato de *type hinting*.

1. `load_database() -> generator`

   Esta función abre el archivo *movies.csv* y retorna el generador con las películas de ese archivo y las demás columnas. Un ejemplo de esta consulta será:

   - `["load_database"]`

2. `filter_by_date(movies:generator, date:int, lower=True:bool) -> generator`

   Esta función recibe un generador con películas y sus demás columnas y retorna un nuevo generador con sólo aquellas películas que tengan año de estreno menor a la señalada en `date` si `lower` es `True`. En el caso contrario, retorna un generador con las películas cuya fecha de publicación sea mayor o igual a `date`. Si se no se introduce el parámetro `lower`, por defecto será `True`. Algunos ejemplos de esta consulta son:

   - `["filter_by_date", ["load_database"], 2015, True]`
   - `["filter_by_date", ["load_database"], 2012, False]`
   - `["filter_by_date", ["load_database"], 2015]`

3. `popular_movies(movies:generator, r_min:int, r_max:int, r_type="All":str) -> generator`

   Retorna todas las **películas y sus demás columnas** de `movies` que tengan un rating entre `r_min` y `r_max` acorde al rating de tipo `r_type`. Si se introduce `"All"`, se debe utilizar el promedio de todos los tipos de *rating*. Si no se introduce el parámetro `r_type`, por defecto será `"All"`. Algunos ejemplos de esta consulta son:

   - `["popular_movie", ["load_database"], 1, 100, "Rotten Tomatoes"]`
   - `["popular_movie", ["load_database"], 50, 100]`
   - `["popular_movie", ["filter_by_date", ["load_database"], 2015, True], 50, 100, "IMDb"]`

4. `best_comments(movies:generator, n:int) -> generator`

   Se debe entregar las `n` películas, y sus demás columnas, con más comentarios positivos, si `n` es un número negativo, entonces entrega las `-n` películas con más comentarios negativos[2]. Algunos ejemplos de esta consulta son:

   - `["best_comments", ["load_database"], 10]`
   - `["best_comments", ["load_database"], -6]`
   - `["best_comments", ["filter_by_date", ["load_database"], 2015, True], 7]`

5. `take_movie_while(movies:generator, column:str, symbol:str, value:int) -> generator`

   Recibe un generador con películas, el nombre de una columna de las películas, algún símbolo de comparación ("`<`", "`>`," "`==`" o "`!=`") y un valor para la comparación. Retornará un nuevo generador con las películas de `movies_generator` **hasta** encontrarse con una película que no cumpla la condición. Las columnas aceptadas son solo: `rt, imdb, metacritic, date o box_office`. Por ejemplo, si tenemos una base de datos así,

Título de la película	IMDb
Spirit	40
El viaje de Chihiro	100
Los vengadores	50
Los Simpsons, la película	60

   ...entonces, algunos ejemplos de consultas y cómo funcionarían son:

[2]Recuerde que si n es negativo, entonces $-n$ es positivo.

- `["take_movie_while", ["load_database"], "IMDb", ">", 30]` retornará las cuatro películas porque todas cumplen con la condición.
- `["take_movie_while", ["load_database"], "IMDb", "<", 50]` retornará solo la primera, porque la segunda ya no cumple.
- `["take_movie_while", ["load_database"], "IMDb", "==", 40]` retornará solo la primera, porque la segunda ya no cumple la condición.
- `["take_movie_while", ["load_database"], "IMDb", "!=", 50]` retornará las primeras dos películas, porque la tercera no cumple la condición y se pierde la última de este filtro aunque cumpla la condición.

2.3.3. Consultas que no retornan otra base de datos

1. `popular_genre(movies:generator, r_type="All":str) -> List[str]`

   Retorna los **cuatro** mejores géneros según `r_type`. Para esto, se calcula el promedio según `r_type` de todas las películas que tenga cada género. Si se introduce `"All"`, se debe utilizar el promedio de todos los tipos de *rating*. Si no se introduce el parámetro `r_type`, por default será `"All"`. Algunas consultas de ejemplo son:

   - `["popular_genre", ["load_database"], "Rotten Tomatoes"]`
   - `["popular_genre", ["load_database"]]`
   - `["popular_genre", ["filter_by_date", ["load_database"], 2015, True], "All"]`

2. `popular_actors(movies:generator, k_actors:int, n_movies: int r_type="All":str) -> List[str]`

   Retorna los k actores más repetidos de las n películas más populares según el tipo de *rating* indicado en `r_type`. Si se introduce `"All"`, se debe utilizar el promedio de todos los tipos de *rating*. Si no se introduce el parámetro `r_type`, por defecto será `"All"`. Se debe entregar a los actores, **en orden** según el número de veces que aparece, con sus películas respectivas. Un ejemplo de *output* sería: "Leonardo Di Caprio, Titanic, Inception, The Revenant". Algunas consultas de ejemplo son:

   - `["popular_actors", ["load_database"], 2, 10, "All"]`
   - `["popular_actors", ["load_database"], 1, 10, "RT"]`
   - `["popular_actors", ["filter_by_date", ["load_database"], 2015, True], 1, 2]`

3. `highest_paid_actors(movies:generator, k_actors=1:int) -> List[str]`

   Retorna a los k actores mejor pagados, junto con cuánto se les paga y, para cada actor, las tres películas que más dinero les haya dado. k siempre debe ser mayor o igual a 1. Puede asumir que el dinero recaudado por las películas se reparte de forma equitativa entre todos sus actores. Si la consulta no recibe `k_actors`, por defecto es 1. Algunos ejemplos de consulta son:

   - `["highest_paid_actors", ["load_database"], 2]`
   - `["highest_paid_actors", ["take_movie_while", ["load_database"], "IMDb", ">", 30], 1]`
   - `["highest_paid_actors", ["filter_by_date", ["load_database"], 2015, True], 1]`

4. `successful_actors(movies:generator) -> List[str]`

   Retorna a todos los actores los cuales todas sus películas hayan tenido los **tres** `rating_type` por sobre el 50 %, lo cual significa que si, dentro de las películas mencionadas en `movies_generator` el actor aparece en alguna que tenga cualquiera de las `rating_type` bajo el 50 %, el actor no deberá aparecer. Algunos ejemplos de consulta son:

- `["successful_actors", ["load_database"]]`
- `["successful_actors", ["take_movie_while", ["load_database"], "IMDb", ">", 30]]`
- `["successful_actors", ["filter_by_date", ["load_database"], 2015, True]]`

Tip: Para resolver consultas anidadas (consultas dentro de otra consulta) una idea es recurrir a la recursión.

2.4. Excepciones

Se espera que se manejen y levanten todos los errores posibles mediante el uso correcto de excepciones. **Una excepción genérica**[3] **no se considerará correcta**, a menos que sea la única solución posible y deberás explicar en el `README.md` por qué es la única solución posible. Debes recordar que se está realizando una librería; por ende, una librería levanta excepciones mientras que los usuarios que utilizan la librería son los que *capturan* los errores. **Sólo se permitirá que su librería *capture* las excepciones definidas en este punto; *capturar* cualquier otra excepción será penalizado.**

Dado que estarás trabajando con películas y actores, deberás crear nuevas excepciones las cuales están detalladas a continuación:

- `BadQuery`

  Esta excepción se genera cada vez que se ingresa una consulta mal escrita o que no existe dentro de las consultas solicitadas y no se puede procesar. Cuando se levanta el error, se debe indicar el nombre de la excepción y de la consulta que generó el error. Ejemplo:

  - `"BadQuery: successful_act"`
  - `"BadQuery: filtrar_por_fecha"`

- `WrongInput`

  Se debe levantar esta excepción si se ingresan mal los parámetros de una consulta. Cuando se levanta el error, se debe indicar el nombre de la excepción, de la consulta donde se generó el error, el parámetro que generó el error y su valor. Ejemplo:

  - `"WrongInput: highest_paid_actors, k_actors, -1"`
  - `"WrongInput: best_comments, movies, 'holi'"`

- `MovieError`

  Se debe levantar esta excepción si durante una consulta se **requiere analizar** un dato de la películas que no está definido. Este podría ser en actores, algún tipo de *rating*, género, fecha de estreno o dinero recaudado. Puede asumir que un dato sin definir se denota con `N/A` en la base de datos. Cuando se levanta el error, se debe indicar el nombre de la excepción, de la consulta donde se generó el error, el nombre de la película que generó el error y el tipo de dato que generó el error. Ejemplo:

  - `"MovieError: successful_actors, Titanic, actors"`
  - `"MovieError: highest_paid_actors, Iron Man, box_office"`

[3] `except Exception`

2.5. *Testing*

En un modulo llamado `testing.py` deberás probar todas las consultas indicadas en la sección 2.3 y todas las excepciones indicadas en la sección 2.4.

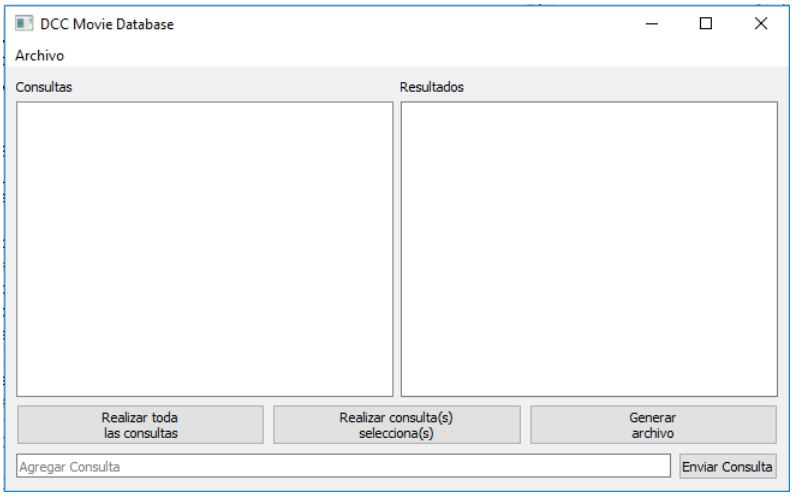
Para *testear* las consultas, todas deben probarse con un caso y en dos de ellas probar dos casos de uso distintos. Para realizar los *tests*, debes utilizar la librería de Python `unittest` sobrescribiendo, de ser necesario, los métodos de `setUp` y `tearDown`. Para realizar el *testing* se podrán crear archivos[4].

Se puede utilizar `for` y `while`; sin embargo, debe cumplir con la filosofía de la sección 5.2.

3. GUI

Para desarrollar el programa, se te entregará una archivo llamado `main.py` con una interfaz, la cual debes utilizar para analizar las consultas. Estas se pueden ingresar directamente utilizando la interfaz o se pueden cargar desde el archivo `consultas.json`. En ambos casos, las consultas serán procesadas sólo si vienen en el formato de listas. Las respuestas deberán escribirse en un archivo llamado `resultados.txt` y se podrá mostrar en la interfaz en paralelo con la consulta asociada cuando se solicite. **No debes modificar la interfaz**; sólo puedes interactuar con ella a través de los métodos que se especifican más adelante.

La interfaz que se les entrega es la siguiente:



Se te entregará el archivo `main.py` que contiene los métodos que deben modificar para trabajar con la interfaz gráfica.
nebCreo que esto está escrito arriba.

3.1. Métodos de la GUI

- `add_answer(text)`: Este método se encarga de agregar cualquier texto al cuadro **Resultados** de la interfaz. Debe recibir un *string* de argumento y **deben** incluir los saltos de línea (`\n`) si quieren poner líneas por separado.

[4]Los archivos deben tener casos de prueba cuyo resultado es conocido.

- `process_query(queries)`: Se llama cuando se presiona el botón "Realizar todas las consultas" o "Realizar consulta(s) seleccionada(s)". Recibe un parámetro llamado `queries`, que es una *lista* con todas las consultas seleccionadas en la interfaz (o todas, en caso de apretar el botón "Realizar todas las consultas").
  Debes modificar este método llamando a tu librería para procesar las consultas y con el método `add_answer` puedes mostrar los resultados de las consultas procesadas en el cuadrado **Resultados** de la interfaz.
- `save_file(queries)`: Se llama cuando se presiona el botón "Generar archivo". Recibe un parámetro llamado `queries`, que corresponde a una lista que contiene a todas las consultas que se han ingresado a la interfaz (es la misma lista que se recibiría en `process_query` si se oprime "Realizar todas las consultas").
  En este método debes implementar todo lo necesario procesar y escribir los resultados en `resultados.txt` respetando el formato entregado. Cada vez que se llama a este método, el archivo se sobrescribe.

4. Archivos relacionados a la tarea

¡Esta sección es importante! No respetar esto afectará considerablemente la nota de su tarea.

4.1. `consultas.json`

JSON (del inglés, ***J**ava**S**cript **O**bject **N**otation*) es un formato estándar de intercambio de datos que puede ser interpretado por muchos lenguajes. JSON es *human-readable*, es decir, puede ser fácilmente leído y entendido por humanos. El formato en que almacena la información es similar a los diccionarios de Python. Para saber más sobre esto, puedes ver el siguiente enlace: `https://json.org/json-es.html`.

Este archivo tendrá el siguiente formato (donde `k` es la cantidad de consultas):

```
[
  [consulta 1],
  [consulta 2],
  ...         ,
  [consulta k]
]
```

Para poder leer este archivo, se debe copiar, **sin cambiar su nombre**, a la misma carpeta que `main.py`. A través de la interfaz, en la barra de opciones, debe seleccionar la opción "Abrir" para cargar el archivo.

Importante: en los archivos `.json` los booleanos se guardan como `true` o `false`, pero al momento de ser cargados en la interfaz, estos serán transformados a la sintaxis de Python: `True` o `False`.

4.2. `resultados.txt`

Este archivo debe indicar como mínimo el número de la consulta y su resultado; es decir, debe tener el siguiente formato:

```
---------- Consulta 1 ----------
RESULTADO CONSULTA 1
---------- Consulta 2 ----------
RESULTADO CONSULTA 2
...
---------- Consulta K ----------
RESULTADO CONSULTA K
```

El formato de cómo muestran el resultado de cada consulta queda a criterio suyo.

4.3. Consultas fallidas, ejecución y ejemplo

Su programa debe:

1. Leer `consultas.json`
2. Generar `resultados.txt` y **no caerse** al encontrar un error durante el procesamiento de alguna consulta, sino indicar el error con el formato que se explicó previamente. La idea es que pueda seguir con las otras. Un ejemplo de manejo de errores a continuación:

```
for i in ["1", 2, 4, "3", 5, 6, "4"]:
    try:
        print(i + 1)
    except TypeError:
        print("Error")
# Se imprime lo siguiente
Error
3
5
Error
6
7
Error
[Finished in 0.1s]
```

Un ejemplo[5] de la lógica de `main.py` para procesar la lista de consultas es:

```
# Abre consultas.json
for query in queries:
    try:
        result = process_query(query)
        # Escribe los resultados en resultados.txt
    except MyException:
        print_exception()
        # Escribe que la consulta falló en resultados.txt e indica el nombre del error
```

```
---------- CONSULTA 1 ----------
Output del error ocurrido
---------- CONSULTA 2 ----------
Output del error ocurrido
---------- CONSULTA 3 ----------
Output del error ocurrido
---------- CONSULTA 4 ----------
Output del error ocurrido
```

Recuerde que cada excepción tiene su propio formato de salida el cual deben respetar en el archivo e interfaz.

5. Consideraciones

5.1. Programación funcional

DCC Movie Database debe estar implementado usando programación funcional. Por lo tanto, sólo se permite el uso de *loops* (`for` y `while`) en:

1. Funciones generadoras
2. Expresiones generadoras

[5] Es solo un ejemplo: pueden escribirlo de la forma que quieran, siempre que se cumpla el procedimiento pedido y el formato mínimo.

3. Contenedores (*e.g.* listas, diccionarios) por comprensión
4. *Loops* que no involucren la interacción con alguna estructura de datos o que retornen algún dato.

A continuación, se mostrarán ejemplos del uso de *loops* permitidos y no permitidos

```
# Este "for" no se puede utilizar.
processed = []
for movie in movies:
    processed.append(process(movie))  # estamos alterando una estructura...

# Este "for" tampoco se puede utilizar si "process" altera la lista.
processed = []
for movie in movies:
    process(movie, processed)  # estamos alterando una estructura...

# Este "for" tampoco se puede utilizar si "process" altera la lista.
for movie in movies:
    processed = process(movie)  # estamos retornando un dato...
    # hacer algo con processed

# En cambio, este "for" sí se puede utilizar si "process" sólo ejecuta algo
# pero no altera una estructura de datos y no está retornando algún dato.
for movie in movies:
    process(movie)

first_elements = (x[0] for x in array_of_arrays)  # Este "for" también está permitido.

# Recuerde que debe utilizar programación funcional para toda la implementación
# del programa como por ejemplo, funciones "lambda" con "map".
double = map(lambda x: x*2, numbers)
```

Si está permitido utilizar `for` y `while` para acceder a cada consulta dentro de la lista de consultas, imprimir los resultados en pantalla y escribir los resultados en el archivo.

```
# Este "for" sí se puede utilizar.
for query in query_list:
    result = process_query(query)
    print(result)

# Esto otro también sería válido.
results = process_queries(query_list)
with open("resultados.txt", "w") as file_:
    for result in results:
        file_.write(result)
```

5.2. Filosofía: *"Do one thing and do it well"*

Dada la similitud en cómo se deben realizar ciertas búsquedas, para esta tarea se aplicará esta filosofía. Por lo tanto, es un requisito que su librería implemente funciones encargadas de hacer sólo una cosa. Esto significa que deben haber funciones pequeñas, que hagan algo específico y corto, que pueda ser generalizado y usado en otras partes del programa. En relación a esto, se evaluará que las funciones definidas no superen un **máximo de 15 líneas**, considerando la definición de la función (`def my_function(*args)`) como una de ellas. Si consideras que alguna función tuya cumple con la filosofía, pero tiene un mayor largo de la cantidad permitida, **debes** justificar en el `README.md` por qué crees que sí cumple con *Do one thing and do it well*.

Deben considerar que una de las ventajas de la programación funcional es la **minimización de efectos colaterales**; es decir, las funciones no modifican variables que no están definidas dentro de la función ni sus *inputs*. Luego, se espera que su librería satisfaga tal ventaja. De no ser posible, **debe justificar en el** `README.md` por qué su función tiene efectos colaterales.

5.3. Uso de clases

Para esta tarea **no se permite el uso de clases**, excepto para crear las excepciones personalizadas, para los casos de prueba en el *testing* o para usar `namedtuples`, si es que lo considera necesario.En caso de necesitar una clase no mencionada antes, deben preguntar en el *issue* de librerías permitidas/prohibidas.

5.4. `.gitignore`

Para no saturar los repositorios de GitHub, **deberás** agregar en tu carpeta `Tareas/T03/` un archivo llamado `.gitignore` de tal forma de **no** subir los archivos de la carpeta `gui` o los CSV de la base de datos. Sin embargo, sí está permitido subir cualquier otro archivo necesario para el *testing*. Además, en caso de tener archivos CSV para el *testing*, esto debe quedar explícito en el nombre. Por ejemplo, `test_database.csv`.

6. Documento entregable

Para esta tarea tendrás que responder un formulario de Google que describa el algoritmo de una función. No es necesario escribir el código, pero sí debe cumplir las siguientes reglas:

- Debe ser una de las siguientes funciones: `highest_paid_actors` o `successful_actors`.
- Debe escribir el pseudocódigo. (clic en el concepto para leer)
- Debe escribir qué herramientas de programación funcional usarás (*e.g.* listas por comprensión, funciones *built-in*, etcétera).

7. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el *Código de honor de la Escuela*: haz clic aquí para leer.
- Tu programa debe ser desarrollado en Python v3.6.
- Tu código debe seguir la guía de estilos descrita en el PEP8.
- Si no se encuentra especificado en el enunciado, asume que el uso de cualquier librería Python está prohibida. Pregunta en el foro si es que es posible utilizar alguna librería en particular.
- El ayudante puede castigar el puntaje[6] de tu tarea, si le parece adecuado. Es recomendable ordenar el código, siendo lo más claro y eficiente posible en la creación de algoritmos.
- Debes adjuntar un archivo `README.md` donde comentes sus alcances y el funcionamiento del sistema (*i.e.* manual de usuario) de forma *concisa* y *clara*. **Tendrás hasta 24 horas después de la fecha de entrega** de la tarea para subir el `README.md` a tu repositorio.
- Crea un módulo para cada conjunto de funciones. Divídelas según el objetivo que cumplan o los tipos de datos que procesan; esto queda a tu criterio. **Se descontará hasta un punto si se entrega la tarea en un solo módulo[7].**
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

Tareas que no cumplan con las restricciones señaladas en este enunciado tendrán la calificación mínima (1.0).

[6] Hasta −5 décimas.
[7] No tomes tu código de un solo módulo para dividirlo en dos; separa el código de forma lógica