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

IIC2233 - Programación Avanzada (I/2017)
Tarea 3

1. Objetivos

- Aplicar conceptos y nociones de programación funcional para el correcto modelamiento de un problema.
- Aplicar conceptos de *testing*.
- Aplicar excepciones.

2. Introducción

Luego de la tragedia mundial que provocaste con las enfermedades, te das cuenta que es muy triste estar solito. Afortunadamente descubres que en el departamento de matemáticas sobrevivieron unas pocas personas. Sin embargo, en un intento de alejarse de los infectados, usaron los computadores de la universidad para hacer una barricada con el resto del campus. Como estabas muy ocupado destruyendo a la humanidad, tu computador quedo intacto, por lo que *Mister Olea* (líder de los sobrevivientes), te indica que debes programar un lenguaje de acceso a bases de datos llamado RQL, el cual podrán usar para diseñar un plan de repoblación de la Tierra.

3. Problema

Para esta tarea, deberás implementar un lenguaje de consultas estadísticas nunca antes visto. Luego, deberás utilizar este lenguaje para responder consultas que ayuden a repoblar la Tierra.

4. El Lenguaje de Consultas RQL

RQL es un lenguaje de acceso a bases de datos creado por *Mister Olea*. RQL permite especificar distintos tipos de consultas en cualquier base de datos y obtener información estadística a partir de ellos como promedio, mediana, desviación, entre otros

4.1. Bases de Datos en RQL

A continuación se mostrará un ejemplo de una posible base de datos soportada por **RQL**, mas adelante se profundizará mas en este tema, esta sección es solo con fines introductorios al lenguaje.

infectados_avistados	muertos_avistados	tiempo_infectado
1594421	1747	7200
492085	839	13800
6068681	610	500
3943144	234	33000
111	146	16500

Figura 1: `EncuestasAnonimas`

La tabla `Encuestas anónimas` nos muestra la cantidad de infectados, la cantidad de gente muerta que ha sido vista por alguien y el tiempo que estuvo infectado. Podemos ver que una persona vió 111 personas infectadas y 146 muertos y otro persona ha visto ... ¡¡1747 muertos!! (pobre).

Todas las bases de datos que ingresaremos tendrán un formato similar a la tabla mostrada anteriormente. Esto permite a **RQL** hacer consultas sobre los datos como: "¿Cuál fue el promedio de gente muerta?", "¿La cantidad de gente infectada es muy dispersa o tienden a ser homogéneos en un punto?", etc. Para poder escribirlas, primero necesitamos definir un lenguaje de consultas.

4.2. Consultas (30 %)

Las consultas son comandos que especifica un conjunto de columnas, tablas y condiciones, y permiten calcular o graficar información a partir de ellas. A continuación se presentan los tipos de datos soportados por **RQL** para luego indicar los comandos reconocidos por este lenguaje, incluyendo los tipos de datos que reciben y los tipos de datos que retornan.Una particularidad de este lenguaje es que el resultado de una consulta puede ser usado como argumento de otra y es capaz de soportar una gran cantidad de estos casos[1].

En la sección 4.4 se muestran algunos ejemplos con la interpretación de que resultado se espera de cada uno.

4.2.1. Tipos de Datos

- `int`, `float`. Representa un número entero, o real.
- `str`. Representa un texto *string*
- `str('A','B','C')`. Es un `str` pero limitado a que debe ser igual a alguno de los elementos mencionados.
- `None`. Representa la ausencia de dato.
- `Funcion`. Representa un dato que puede ser *callable*, es decir, se le puede entregar argumentos y retornará algún resultado.
- `Columna`. Representa un conjunto de datos numéricos. Puede ser una lista, un *set* o cualquier otro que pueda contener datos numéricos. Debe ser iterable.
- `Consulta`. Representa una lista que contiene a un comando con sus respectivos argumentos. Por ejemplo: `['extraer_columna', 'avanzada', 'notas_T01']`
- `any`. Puede ser cualquier tipo de dato mencionado anteriormente.

[1] EL ultimo registro indica que la consulta mas grande tenía mas de 1000 consultas dentro usada como argumento para diversas operación

4.2.2. Comandos básicos

- `asignar(variable, comando o dato)`:

```
:param variable: str
:param comando o dato: any
:returns: None
```

  Guarda el valor de retorno de `comando`, o el valor de `dato`, bajo el nombre `variable`. asignar no permite guardar información si el nombre ingresado es igual al nombre de algún comando propio de RQL o al nombre de alguna distribución de probabilidad.

- `crear_funcion(nombre_modelo, *parámetros)`:

```
:param nombre_modelo: str ("normal", "gamma", "exponencial")
:param *parametros: int or float
:returns: funcion
```

  Recibe el nombre de algún modelo de probabilidad (sección 4.3) y los `parámetros` necesarios por el modelo, y retorna una nueva función que puede ser ocupada en el comando `evaluar` para generar una columna de datos

- `graficar(columna, opción)`:

```
:param columna: Columna
:param opcion: str or Columna
:returns: None
```

  Grafica (2D) los datos de una columna. Para esta implementación se debe utilizar `matplotlib 2.0` y **únicamente** `numpy 1.12`. El eje Y corresponderá a los valores de `columna`. El eje X dependerá de `opción`:

  `'numerico'` El eje X corresponde a una enumeración de los datos. Por ejemplo, si los datos son `[10, 12, 13]`, entonces los puntos del gráfico serán `[(0, 10), (1, 12), (2, 12)]`

  `'normalizado'` El eje X corresponde a una enumeración normalizada de los datos. Por ejemplo, si los datos son `[10, 12, 13]`, los puntos del gráfico serán `[(0/(10+12+13), 10), (1/35, 12), (2/35, 12)]`

  `'rango:a,b,c'` Dado un string con el formato `'rango:a,b,c'`, se deberá interpretar del siguiente modo: Se define un rango de valores dado `[a, b]` y un *step* `c`, el eje X se compondrá por los números generados por dicho rango. Esto solo es posible si la cantidad de números generados por el rango es mayor o igual a la cantidad de datos. Además, si `a<b`, entonces `c` debe ser positivo; mientras que si `a>b`, entonces `c` debe ser negativo.

  `Columna` Si, en lugar de recibir un *string*, se recibe otra columna de datos en el argumento `opcion`, los valores de la nueva columna serán el eje X. Esta columna **debe** tener la misma cantidad de datos que la utilizada para el eje Y.

- **Símbolos**. El lenguaje permite usar dos tipos de símbolos.
  - Símbolos para comparaciones: `==`, `>`, `<`, `>=`, `<=`, y `!=`
  - Símbolos para operaciones:`+`, `-`, `*`, `/`, y `>=<` (aproximación).

4.2.3. Comandos que retornan un conjunto de datos

- `extraer_columna(nombre_de_archivo, columna)`:

```
:param nombre_del_archivo: str
:param columna: str
:returns: Columna
```

Dado un `nombre_de_archivo` en formato `csv`, retorna la columna cuyo nombre (ó *head*) sea igual a `columna`.

- `filtrar(columna, símbolo, valor)`:

```
:param columna: Columna
:param simbolo: str
:param valor: int or float
:returns: Columna
```

Dado un conjunto de datos en `columna`, retorna todos los que pasen el filtro dado por `símbolo` y `valor`. El argumento `simbolo` debe ser un simbolo de comparación.

- `operar(columna, simbolo, valor)`:

```
:param columna: Columna
:param simbolo: str
:param valor: int or float
:returns: Columna
```

Dado un conjunto de datos, un `simbolo` de operación, y un `valor`, se aplica la operación indicada por `simbolo` y `valor` a cada dato de `columna`, y se retornan esos resultados. Si el `símbolo` es `>=<`, entonces `valor` debe ser ≥ 0, e indica cuántos decimales aproximar. El argumento `símbolo` debe ser un símbolo de operación.

- `evaluar(función, inicio, final, intervalo)`:

```
:param funcion: funcion
:param inicio: int or float
:param final: int or float
:param intervalo: int or float
:returns: Columna
```

Dada una `función` creada por el comando `crear_funcion`, se evalúa `función` desde `inicio` hasta `final` a intervalos de `intervalo`. Retorna los resultados.

4.2.4. Comandos que retornan un valor numérico

Dado un conjunto de datos numéricos $X = \{X_0, \ldots, X_{n-1}\}$, los siguientes comandos reducen el conjunto a un único dato numérico (`int` ó `float`). El formato es:

`[comando, datos]`,

que indica que se debe aplicar `comando` al conjunto de datos indicado por `datos`. Por ejemplo,

`["PROM", [1,2,3,4,5,5,6]]`

retornará el promedio de la lista `[1,2,3,4,5,5,6]`.

```
:param datos: Columna
:returns: int or float
```

- `LEN`. Retorna la cantidad de datos en X.
- `PROM`. Retorna el promedio aritmético de los datos en X.

$$\texttt{PROM} = \overline{X} = \frac{\sum_{i=0}^{n-1} X_{i-1}}{n}$$

- `DESV`. Retorna la desviación estándar de los datos en X.

$$\texttt{DESV} = \sigma = \sqrt{\frac{\sum_{i=0}^{n}(X_i - \overline{X})^2}{n-1}}$$

- `MEDIAN`. Retorna la mediana de los datos en X.

  Corresponde al valor que se encuentra en la mitad de los datos cuando están ordenados de menor a mayor. Si se tiene un número par de datos, la mediana se calcula como el promedio de los dos valores que se encuentran al medio. Puede suponer que al solicitar la mediana, la columna ya estará ordenada.

- `VAR`. Retorna la varianza de los datos en X.

$$\texttt{VAR} = \sigma^2 = \frac{\sum_{i=0}^{n}(X_i - \overline{X})^2}{n-1}$$

4.2.5. Comandos que retornan un booleano

Para todos estos comandos, el símbolo a utilizar será un símbolo de comparación.

- `comparar_columna(columna_1, simbolo, comando, columna_2)`:

```
:param columna_1: Columna
:param simbolo: str
:param comando: str ("LEN", "PROM", "DESV", "MEDIAN", "VAR", "KUR", "ASI", "VAR-COEF", "KUR-COEF" or "ASI-COEF")
:param columna_2: Columna
:returns: any
```

  Recibe las columnas de datos `columna1` y `columna2`; aplica `comando` a ambas y retorna la comparación de ambas de acuerdo a `simbolo`.

- `comparar(numero_1, simbolo, numero_2)`:

```
:param numero_1: int or float
:param simbolo: str
:param numero_2: int or float
:returns: any
```

  Retorna la comparación entre `numero_1` y `numero_2` de acuerdo a `simbolo`.

4.2.6. Comandos compuestos

- `do_if(consulta_a, consulta_b, consulta_c)`:

```
:param consulta_a: Consulta
:param consulta_b: Consulta_que_retorna_un_bool
:param consulta_c: Consulta
:returns: any
```

  Si `consulta_b` es `true`, realiza el `consulta_a`, en caso contrario, realiza el `consulta_c`.

4.3. Distribuciones de probabilidad (8 %)

El comando `crear_fun cion` recibe como argumento uno de los siguientes modelos junto a los parámetros necesarios para que quede como una función de un valor `x`. Por ejemplo, el modelo `normal` recibe dos parámetros (μ y σ); por lo tanto, para crear una función normal con $\mu = 0$ y $\sigma = 1$, se tendría que escribir el siguiente comando:

```
['crear_funcion', 'normal', 0, 1]
```

Esa consulta creará una función normal donde $\mu = 0$ y $\sigma = 1$. Luego, con el comando `evaluar`, se evaluará x en el rango señalado para crear una columna de datos que se puede analizar.

A continuación se definen las distribuciones admitidas por **RQL** junto con los parámetros necesarios para generar estas funciones que solo dependerán de x.

- `normal`(μ, σ)

$$f_X(x) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2\right], -\infty < x < \infty$$

  Con μ parámetro de localización y σ un parámetro de escala o forma tales que $\infty < \mu < \infty, 0 < \sigma < \infty$

- `exponencial(`ν`)`

$$f_X(x) = (\nu)e^{-\nu x}, \quad x >= 0$$

  Con $\nu > 0$

- `gamma (`ν`, `k`)`

$$f_X(x) = \frac{(\nu)^k}{(k-1)!} x^{k-1} e^{-\nu x}, \quad x >= 0$$

4.4. Ejemplos Comandos

- `['asignar', 'x', ['extraer_columna', 'avanzada', 'notas_T01']]`

  Abre el archivo `avanzada.csv`, extrae la columna llamada `notas_T01`, y la guarda bajo el nombre de variable `x`.

- `['comparar', ['PROM', 'x'], '>', ['DESV', 'x']]`

  Calcula el promedio de los datos en la variable `x` y su desviación estándar. Retorna el resultado de la comparación "promedio mayor que desviación".

- `['asignar', 'funcion_normal', ['evaluar', ['crear_funcion', 'normal', 0, 0.5], -3, 5, 0.1]]`

  Crea una función usando el modelo `normal` con $\mu = 0$, $\sigma = 0{,}5$. Luego evalúa esta función en el rango $-3, 5$ con intervalo 0,1, y guarda ese conjunto de datos bajo la variable `funcion_normal`.

- `['graficar', 'funcion_normal', 'normalizado']`

  Grafica los datos guardados bajo variable de nombre `funcion_normal` usando la opción `normalizado`.

- `['do_if', ['VAR', 'funcion_normal'], ['comparar_columna', 'funcion_normal', '>', 'DESV', 'x'], ['PROM', 'x']]`

  Compara la desviación estándar de la normal y de `'x'`. Si la desviación de la normal es mayor a la desviación de `'x'`, se calcula la variación de la normal, en caso contrario, se calcula el promedio de `'x'`.

5. Manejo de errores (12 %)

Su programa debe poder manejar todo error que se genere a partir del uso de los comandos, los cuales deben seguir el siguiente formato:

```
Error de consulta: Comando Error
Causa: Error
```

Donde *Error de consulta* y *Causa* son strings fijos; 'Comando Error' es un string con el comando especifico que causó el error junto a sus argumentos y 'Error' es el nombre del error generado.

Si uno de los argumentos es otra consulta, no deben escribir toda la consulta contenida, sino solo el nombre del comando de dicha consulta seguido de un *.

Ejemplo

```
['filtrar','x', '>', ['PROM',['extraer_columna', 'notas_avanzada', 'Tarea_1'] ] ]
```

Error de consulta: `filtrar('x', '>', 'PROM'*)`
Causa: `Referencia invalida`

Errores posibles

- **Argumento inválido**. El comando recibe una cantidad diferente de argumentos a lo esperado.

  `['asignar', 'x', 1, 2, 3]`

  `['graficar', [1,2,3]]`

- **Referencia inválida**. Se hace referencia a una variable no definida

  Ejecutar `['filtrar','x', '<', [PROM,'x']]` antes de definir `x`

- **Error de tipo**. Se entrega un argumento distinto a lo esperado.

  `['extraer_columna', 2, 'poblacion']`

  `['amplificar', 'casa', 5]`

- `Error matemático`. Alguna operación está dividiendo por 0

  `['VAR-COEF', [-2,-1,0,1,2]]`

  La formula pide dividir por el promedio el cual es 0

- **Imposible procesar**. El argumento entregado no cumple los requisitos del comando

  `['graficar', [1,2,3], [1,2]]`

  `['crear_funcion','exponencial',0]`

  `['extraer_columna', 'paises', 'notas']`, si `paises.csv` no existe, o no tiene la columna `notas`

  Recuerde usar excepciones donde corresponda para identificar varios de estos errores.

  Para ayudar a su *testing*, puede considerar que ante cualquier error, el comando retornará `None`.

  En caso de aparecer un nuevo error que no está considerado en los anteriores, pueden manejarlo mediante una nueva excepción o incluirlo en alguna de las anteriores pero siempre especificando en el Readme su decisión.

6. Bases de Datos

Usted dispondrá de archivos `csv` (*comma separated values*) para describir las bases de datos. Estos podrán ser de cualquier temática, y deberá poder cargarlos en su programa tratando de ser lo más eficiente en memoria, ya que podrían ser de gran tamaño (y quedarse sin RAM en el proceso).

6.1. `CSV`

Los archivos CSV son archivos de texto que permiten guardar tablas de la base de datos de acuerdo al formato:

```
Nombre: str, Sección: int, Nota_T01: float, Nota_T02: float
Fernando, 2, 6.7, 1.2
Felipe, 2, 7.5, 6.8
Juan, 1, 3.3, 5.6
```

El nombre del archivo corresponde al nombre de la tabla. Por ejemplo, el archivo `alumnos.csv` contiene los datos de la tabla `alumnos`. La primera línea, llamada *header*, contiene los nombres (*head*) de las columnas seguido del tipo de dato. Las lineas que siguen son los registros (tuplas). Cada línea contiene un dato por cada columna, separado con una coma (`,`).

Un CSV se encuentra bien formateado cuando tiene la misma cantidad de columnas en todas sus líneas. **Puede suponer que el intérprete de RQL solo recibirá archivos `.csv` bien formateados.**

7. Consideraciones (30 %)

Estas consideraciones se evalúan transversalmente durante todo el programa.

7.1. Programación Funcional

El intérprete de RQL debe estar implementado usando programación funcional. Se permite el uso de *loops* (`for` y `while`) **solo en generadores por comprensión**. A continuación, se mostrarán ejemplos del uso de loops no permitidos y permitidos[2]:

```
# Este for no se puede utilizar
for empleo in empleos:
        dame_comida(empleo)

#En cambio, este for si se puede utilizar
primer_elemento = (x[1] for x in listas_de_listas)

# Recuerde que debe utilizar programación funcional para toda la implementación de
# RQL,como por ejemplo funciones lambda y map:
dobles = map(lambda x: x*2, numeros)
```

7.2. Funciones Circulares

NO se permite el uso de funciones circulares, es decir, si una función `f` llama a una función `g`, esta última no puede llamar a `f`. Ejemplo:

```
 # ¡¡Esto no se puede hacer!!

consulta = ["Prom", ["extraer_columna", "base", "infectados_avistados"]]

def analizar_consulta(consulta):
        if consulta[0] == "Prom":
            return promedio(consulta[1])
    elif consulta[0] == "extraer_columna":
        --otras cosas --
           ...
```

[2] Nos interesa que no usen los ciclos `for` y `while` en lo que son las funcionalidades lógicas del programa. Sí pueden utilizarlos, por ejemplo, para imprimir valores.

```python
def promedio(datos):
        if isinstance(datos, list):
          #Lo que se quiere calcular es otra consulta
          consulta = analizar_consulta(datos) # Esto está mal
          -- calcular promedio--
        else:
          --calcular promedio--
```

Sí se permite que una función se llame a si misma.

7.3. Uso de clases

Para esta tarea **no se permite el uso de clases para la implementación de RQL**, solo para las excepciones personalizadas y el testing si lo considera necesario.

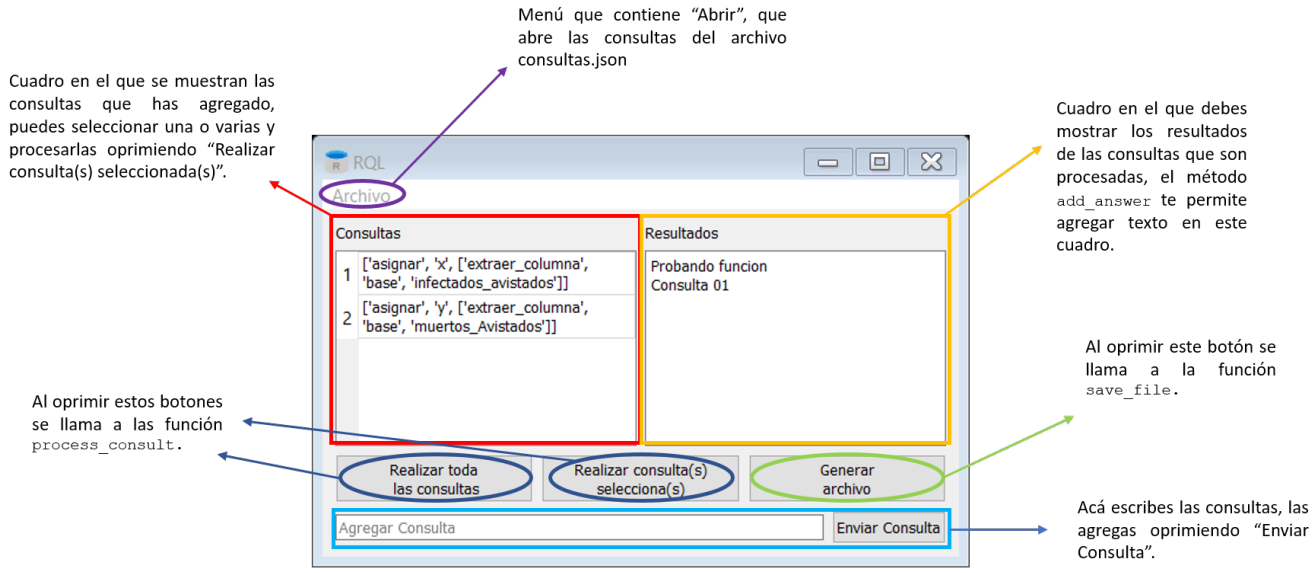
8. Archivos relacionados a la tarea

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

8.1. `main.py`

Se les entregará una archivo llamado `main.py` con una interfaz el cual deben modificar para analizar las consultas que son ingresadas en la interfaz directamente o aquellas que se encuentran en el archivo `consultas.json`, las respuestas a estas deberán ser escritas en `resultados.txt` cuando se solicite. **No se pide que modifiques la interfaz**, solo que interactúes con ella a través de los métodos que se especifican más adelante.

La interfaz que se les entrega es la siguiente:



El archivo `main.py` contiene los métodos que deben modificar para trabajar con la interfaz gráfica en la que se harán las consultas y se mostrarán los resultados. Los métodos a modificar son:

- `process_consult(self, querry_array)`. Es llamado cuando presionas los botones "Realizar todas las consultas" o "Realizar consulta(s) seleccionada(s)". Recibe `querry_array`, que es una lista que contiene las consultas seleccionadas en la interfaz (o todas, en caso de apretar el botón "Realizar todas las consultas–).
  Debes modificar este método con tus funciones para procesar las consultas. Con el método `add_answer` puedes mostrar los resultados de las consultas procesadas en el cuadrado **Resultados** de la interfaz.

- `save_file(self, querry_array)`. Es llamado cuando presionas el botón "Generar archivo–. Recibe `querry_array`, 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_consult 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.

8.2. `consultas.json`

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.

8.3. `resultados.txt` (5 %)

Este archivo debe indicar como mínimo el numero de 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
```

8.4. 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 explico previamente. La idea es que pueda seguir con las otras. En caso de que una consulta solicite graficar o asignar, solo debe dejar la palabra `grafica` o `asignar` como resultado de la consulta. 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[3] de la lógica de main.py a continuación:

```
if __name__ == "__main__":
    # Abre consultas.txt
    for i in consultas:
        try:
            procesar_consulta(i)
            # Escribe los resultados en resultados.txt
        except MyExcepcion:
            imprimir_fallo()
            # Escribe que la consulta falló en resultados.txt y la causa
```

De esta forma, el archivo `resultados.txt` debería verse como:

```
-------- CONSULTA 1 ----------
asignar
------- CONSULTA 2 ----------
Error de consulta: filtrar(x, >, 'PROM'*)
Causa: Referencia invalida
------- CONSULTA 3 ----------
100,2
------- CONSULTA 4 ----------
grafica
```

Junto al enunciado hay algunos archivos de ejemplo con algunas consultas y sus resultados. Además se le adjuntó el tiempo que se demoró cada consulta en realizarse para que tengan una idea de cuanto tiempo debería demorarse sus funciones. **Es importante que su programa funcione con cualquier tipo de archivo (siempre que mantengan la estructura detallada anteriormente, por ejemplo que los csv tengan en todas sus filas la misma cantidad de columnas, y que en cada columna los datos sean del mismo tipo). No lo adapten a los archivos de ejemplo.**

9. Excepciones

Se espera que se manejen todos los errores posibles mediante el uso correcto de excepciones (i.e: indicando el tipo de excepción a capturar y lo que se hará con ella. Una excepción genérica no se considerará como 100 % correcta, a menos que sea la única solución posible - explicar en el README.md por qué es la única solución posible).

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

10. Testing (15 %)

Debes testear las funciones más importantes de tu programa, como mínimo debes tener testeadas las funciones que realicen los siguientes comandos: asignar_variable, filtrar, evaluar, PROM, MEDIAN, VAR, DESV, comparar_columna y do_if. Los test que crees deben ser ejecutados sobre el lanzamiento de excepciones (deben verificar que efectivamente si ocurre un error, se arroje la excepción correspondiente a ese error), además deben verificar que la función que se testea tenga el funcionamiento esperado (si la función busca calcular el promedio de un set de datos, que efectivamente el promedio que calcula sea correcto).**Ademas debes testear los 5 tipos de errores distintos**.
Importante: Todos los test deben estar escritos en un archivo llamado my_tests.py

11. Restricciones y alcances

- Desarrolla el programa en Python 3.5
- La tarea es individual, y está regida por el Código de Honor de la Escuela: Click para Leer.
- Su código debe seguir la guía de estilos `PEP8`
- Si no se encuentra especificado en el enunciado, asuma que el uso de cualquier librería Python está prohibido. Pregunte en el foro si se pueden usar librerías específicas.
- El ayudante puede descontar el puntaje de hasta 5 décimas de tu tarea, si le parece adecuado. Recomendamos ordenar el código y ser lo más claro y eficiente posible en la creación de los algoritmos.
- Adjunte un archivo `README.md` donde comente sus alcances y el funcionamiento de su sistema (*i.e.* manual de usuario) de forma *concisa* y *clara*. Tiene hasta 24 horas después de la fecha de entrega de la tarea para subir el `README.md` a su repositorio.
- Separe su código de forma inteligente, estructurándola en distintos módulos. Divídalas por las relaciones y los tipos que poseen en común. **Se descontará hasta un punto si se entrega la tarea en un solo módulo.**
- Cualquier detalle no especificado queda a su criterio, siempre que no pase por sobre los requerimiento definidos en el enunciado.

12. Entrega

Código

- **Fecha/hora:** viernes 5 de mayo del 2017, 23:59 horas.
- **Lugar:** GIT - Carpeta: Tareas/T03

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