Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación

IIC2233 Programación Avanzada (2025-1)

Tarea 3

Entrega

- **Tarea y** `README.md`
  - **Fecha y hora oficial (sin atraso):** viernes 30 de mayo de 2025, 20:00.
  - **Fecha y hora máxima (2 días de atraso):** domingo 1 de junio de 2025, 20:00.
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T3/`.
    El código debe estar en la rama (*branch*) por defecto del repositorio: `main`.
  - **Pauta de corrección**: en este enlace.
  - **Bases generales de tareas (descuentos)**: en este enlace.
  - **Formulario entrega atrasada:** en este enlace. Se cerrará el domingo 1 de junio, 23:59.
- **Ejecución de tarea**: La tarea será ejecutada **únicamente** desde la terminal del computador. Además, durante el proceso de corrección, se cambiará el nombre de la carpeta "`T3/`" por otro nombre y se ubicará la terminal dentro de dicha carpeta antes de ejecutar la tarea.

Objetivos

- Entender y aplicar el paradigma de programación funcional para resolver un problema.
- Manejar datos de forma eficiente utilizando herramientas de programación funcional:
  - Uso de generadores y funciones generadoras.
  - Uso de `map`, `lambda`, `filter`, `reduce`, etc.
  - Uso de estructuras por comprensión.
  - Uso e investigación de las librerías `itertools` y `collections`.
- Utilizar conceptos de interfaces y `PyQt5` para implementar una aplicación gráfica e interactiva. Además en esta se debe entender y aplicar los conceptos de *back-end* y *front-end*.
- Aplicar conocimientos de señales.

Índice

1. *DCC*

Luego de haber dedicado gran parte de tu tiempo a la poda y simulación de árboles, decides embarcarte en una nueva aventura; algo más distinto, para ampliar tus conocimientos. Es por esto que durante esta búsqueda te topas con el gran *DCC*, el **Departamento de Consultas al Comercio**. Este gran lugar se caracteriza por recibir una gran cantidad de datos sobre distintos comercios y poder realizar una serie de consultas a ellos para luego visualizarlos en una interfaz gráfica y así mostrarle dicha información a sus clientes respectivos. Ahora bien el *DCC* se caracteriza por brindar un servicio personalizado a cada uno de sus clientes, por lo que cada uno de ellos recibe una solución totalmente personalizada y adecuada a sus requisitos.

Como programador entusiasmado decides ser parte de uno de estos desafíos y dedicarte a brindarle la mejor solución al nuevo cliente que ha mandado su gran cantidad de datos. El programa que el cliente ha solicitado consiste en un **motor de consultas**, el cual debe interactuar con los datos del cliente. Este motor tiene como objetivo permitir al cliente seleccionar alguna de las consultas del conjunto de consultas que ha pedido que sean desarrolladas y, luego de ejecutarla, poder visualizar el resultado de ella. Ahora bien, el cliente también ha pedido que dicho motor presente una pequeña ventana de bienvenida antes de mostrar el programa en sí.

Uno de los requisitos finales del cliente es poder obtener los resultados de estas consultas en un **tiempo óptimo**. Es por esto que el programa que le entregues debe ser capaz de generar la respuesta a cada consulta en no más del tiempo que el cliente encuentre adecuado.

2. Flujo del programa

En *DCC*, el principal objetivo será poder realizar consultas en torno a distintas bases de datos que tendrás disponibles. Esta tarea comenzará con funciones para acceder a archivos de distintos tamaños. Tu objetivo en esta sección será almacenar la información de los archivos en `namedtuples` para poder utilizarla más adelante. Luego, utilizarás tus habilidades de programación funcional y generadores para realizar consultas, las cuales darán información específica sobre las distintas órdenes, productos, usuarios o proveedores. Por último, utilizando los contenidos de interfaces gráficas, deberás crear una ventana de bienvenida junto con una interfaz para poder seleccionar consultas y visualizar sus resultados.

La realización de la tarea se dividirá en dos partes principales. En primer lugar, se pedirá que implementes las funciones mencionadas más adelante. Para que el código sea ordenado, estas funciones ya están declaradas en los archivos que te entregamos. Es muy importante que **no le cambies el nombre a la función y que no modifiques los argumentos recibidos**. Además de los archivos entregados por nosotros, está permitido crear nuevos archivos y/o crear otras funciones si lo estimas conveniente. Esta sección será evaluada mediante el uso de tests. Para poder asegurarse de la buena realización de las funciones, se podrá utilizar una serie de tests públicos. Es sumamente importante que **no realices la tarea basándote en los tests públicos**. Te debes basar en el enunciado y sólo puedes usar los tests públicos para apoyo. Debido a esto, es importante destacar que **si el alumno tiene los test públicos correctos, pero los tests privados incorrectos, no se otorgará puntaje**. Se explica cómo manipular los *tests* en la Subsección 5.1: Ejecución de *tests*.

La segunda sección de la tarea será la parte visual del programa, el cual será desarrollado utilizando los contenidos de interfaces gráficas aprendidos en el curso. Deberás implementar dos ventanas: La primera ventana consiste en una ventana de inicio, y la segunda ventana será la encargada de realizar y visualizar el resultado de las consultas. Esta parte de la tarea será corregida manualmente, por lo que es sumamente importante que **el código esté ordenado para facilitar el entendimiento**.

Aunque -como se menciono anteriormente- puedes crear los archivos que quieras, en el directorio de la tarea se encuentran los siguientes archivos y directorios como base:

Modificar `main.py`: Archivo, inicialmente vacío, donde se instancian y conectan los distintos componentes del *frontend* y *backend*, para así ejecutar el programa.

Modificar `backend/`: Directorio que contiene todo lo asociado al *backend* del programa. Adicionalmente, contiene el siguiente archivo:

- Modificar `consultas.py`: Archivo que contiene las consultas del programa. Las funciones a implementar se explican en la Sección 3: Programación Funcional.

Modificar `frontend/`: Directorio que contiene todo lo asociado al *frontend* del programa, es decir, todo lo relacionado a la interfaz gráfica.

No modificar `data/`: Directorio que contiene una serie de archivos CSV necesarios para ejecutar la tarea. Dentro de esta carpeta habrá tres subcarpetas (S, M, L) que contendrán los CSV de distintos tamaños. En un principio esta carpeta se encontrará vacía, debido a que los distintos archivos se deben descargar y ordenar desde **este link**.

No modificar `tests_publicos/`: Directorio que contiene los tests públicos de la tarea. Esto se encargarán de revisar lo implementado en `consultas.py`. En un principio esta carpeta se encontrará vacía, debido a que los distintos archivos se deben descargar y ordenar desde **este link**.

Importante: los *tests* entregados en esta carpeta no serán los mismos que se utilizarán para la corrección de la evaluación.

No modificar `utilidades.pyc`: Archivo que contiene las `namedtuples` a utilizar, junto a funciones útiles para el desarrollo de la tarea. Este archivo ya se encuentra implementado.

Los archivos y directorios listados anteriormente se deberán organizar de la siguiente manera dentro de la carpeta `T3/` una vez que se haya descargado toda la información necesaria:

```
T3/
├── backend/
│   ├── consultas.py
│   └── ...
├── frontend/
│   └── ...
├── data/
│   ├── S/
│   ├── M/
│   └── L/
├── tests_publicos/
│   ├── solution/
│   │   ├── test_01.py/
│   │   ├── test_02.py/
│   │   └── ...
│   ├── test_00_cargar_datos.py/
│   ├── test_01_productos_despues_fecha_carga_datos.py/
│   └── ...
├── utilidades.pyc
├── main.py
├── README.md
└── .gitignore
```

3. Programación Funcional

Para poder analizar a la gran cantidad de datos que posee el cliente de *DCC*, se necesitará de tu ayuda experta para obtener información mediante la realización de diversas consultas, por lo que deberás aplicar tus conocimientos de programación funcional.

Para esto, deberás interactuar con distintos tipos de datos, los que serán explicados con mayor detalle en la Subsección 3.1: Datos y completar distintas funciones pedidas en la Subsección 3.2: Carga de datos y la Subsección 3.3: Consultas, que se realizará mediante la modificación de un código pre-existente.

3.1. Datos

Para poder interactuar con los datos entregados por los clientes, tendrás que leer los archivos CSV y procesar su contenido. Para poder manejar el contenido de estos archivos se te entregan las `namedtuples` del archivo `utilidades.pyc`.

A modo general, los datos pueden presentar atributos que seguirán un formato en específico:

- **Id:** Los datos contarán con identificadores únicos (id) que permitirán distinguir y asociar los datos. El formato de estos ids consitirán en un *string* que pueden contener caracteres alfanuméricos y guiones (`-`, `_`).
- **Fecha:** Cada atributo que haga relación a una fecha consistirá en un *string* con el siguiente formato `"YYYY-MM-DD"` (`AÑO-MES-DÍA`).
- **Dirección:** Cada atributo que haga relación a una dirección utilizará una adaptación del formato para enviar paquetes detallado en el siguiente **enlace** bajo el titulo "Cómo escribir una dirección postal de EE. UU.".

  Esta adaptación **NO POSEE** el nombre del destinatario en sus detalles y posee toda su información en **una sola linea** separada por espacios. Esto quiere decir que una dirección siempre seguirá el siguiente formato:

  $$\textit{dirección} = \textit{calle ciudad estado código\_zip}$$

  donde, *calle* y *ciudad* pueden contener varias palabras y números.

A continuación se detallan los distintos tipos de datos y sus respectivas `namedtuples`:

3.1.1. Usuarios

Indica la información de un usuario. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
`id_base_datos`	`str`	Permite distinguir e identificar a cada usuario.	`"d09f76b2-39be-4408-a871"`
`nombre_completo`	`str`	Nombre y apellido del usuario, separado por un espacio.	`"Jacqueline Smith"`
`correo`	`str`	Correo electrónico del usuario.	`"jacquelinesmith@calderon.com"`
`fecha_nacimiento`	`str`	Fecha de nacimiento del usuario.	`"1963-12-28"`
`direccion`	`str`	Dirección del usuario.	`"West Kennethside PR 14580"`
`codigo_zip`	`str`	Código postal de la dirección del usuario.	`"9402"`
`numero_telefonico`	`str`	Número de teléfono del usuario.	`"+1-600-984-6136x7182"`

3.1.2. Ordenes

Indica la información que contiene una orden. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
`id_base_datos`	`str`	Permite distinguir e identificar una orden en el archivo.	`"41317706-46db-4ca4-af3a"`
`id_base_datos_usuario`	`str`	Permite distinguir e identificar a que usuario corresponde la orden.	`"d816e625-d82b-431a-9b29"`
`estado_orden`	`str`	Representa el estado de la orden (*shipped*, *pending*, *delivered*, *cancelled*).	`"shipped"`
`fecha_creacion`	`str`	Fecha de creación de la orden.	`"2024-10-13"`

3.1.3. OrdenesItems

Indica los productos que contiene una orden. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
`id_base_datos_orden`	`str`	Permite distinguir e identificar una orden en el archivo.	`"41317706-46db-4ca4-af3a"`
`id_base_datos_producto`	`str`	Permite distinguir e identificar un producto dentro de la orden.	`"5420e0c0-d33b-41f2-9ab0"`
`cantidad_productos`	`int`	Entero que representa la cantidad de productos comprados en dicha orden.	`5`

3.1.4. Productos

Indica la información de un producto. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
`id_base_datos`	`str`	Permite distinguir e identificar un producto dentro de la base de datos.	`"4b1f68bb-419b1"`
`nombre`	`str`	Nombre del producto.	`"COOLER ICE WINE LINEA"`
`precio`	`float`	Precio del producto.	`168.5091`
`cantidad_por_unidad`	`int`	Entero que indica la cantidad de productos que se venden por unidad de medida.	`6`
`unidad_de_medida`	`str`	Unidad de medida de cada producto. Esta puede ser EA (*each*), PK (*package*), o CS (*case*).	`"PK"`
`categoria`	`str`	Categoría a la que pertenece el producto.	`"Ice Buckets"`
`identificador_del_proveedor`	`str`	Identificador único utilizado por el proveedor para identificar un producto.	`"56540-24"`
`fecha_modificacion`	`str`	Fecha de modificación del producto.	`"2024-07-20"`

Para clarificar la relación entre los atributos `cantidad_por_unidad` y `unidad_de_medida` se presenta el siguiente ejemplo: Si un producto corresponde a una *six pack* de cervezas, el cual contiene 6 cervezas, entonces el atributo `cantidad_por_unidad` valdría `6`, mientras que el atributo `unidad_de_medida` sería `"PK"`.

3.1.5. Proveedor

Indica la información de un proveedor. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
`nombre_proveedor`	`str`	Nombre del proveedor.	`"Perez, Lopez and West"`
`estado`	`str`	Estado del proveedor.	`"VI"`
`direccion`	`str`	Dirección del proveedor.	`"812 May Hunterside SC 56209"`
`codigo_zip`	`str`	Código postal del proveedor.	`"56209"`
`numero_telefonico`	`str`	Número de teléfono del proveedor. No presenta un formato en especifico.	`"638-221-3454x742"`
`correo`	`str`	Correo electrónico del proveedor.	`"info@mullenllc.com"`

3.1.6. ProveedoresProductos

Indica la relación entre un producto y un proveedor. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
`nombre_proveedor`	`str`	Nombre del proveedor.	`"Perez, Lopez and West"`
`identificador_del_proveedor`	`str`	Identificador único utilizado por el proveedor para identificar un producto.	`"56540-24"`

3.2. Carga de datos

En la carpeta `data/` podrás encontrar tres subcarpetas (`S`, `M` y `L`) con bases de datos de distintos tamaños: pequeños, medianos y grandes, respectivamente. Deberás copiar esta carpeta y ubicarla en tu carpeta de la tarea `T3/`, siguiendo la estructura indicada en Sección 2: Flujo del programa.

Para poder trabajar las consultas de esta tarea deberás cargar la información en **generadores** que contengan las `namedtuple` definidas en `utilidades.pyc`. La información de estos **generadores** se obtendrá a partir de archivos de extensión `.csv` que siguen el mismo formato de las `namedtuple` mencionadas en la sección anterior.

Para realizar la carga de información a generadores deberás completar las siguientes funciones:

Modificar `def cargar_usuarios(path: str) -> Generator:`

Recibe el `str` `path` correspondiente a la ruta del archivo que contiene la información de los **usuarios**. Retorna un **generador** con instancias de `namedtuple` `Usuarios` asociadas al archivo.

Modificar `def cargar_productos(path: str) -> Generator:`

Recibe el `str` `path` correspondiente a la ruta del archivo que contiene la información de los **productos**. Retorna un **generador** con instancias de `namedtuple` `Productos` asociadas al archivo.

Modificar `def cargar_ordenes(path: str) -> Generator:`

Recibe el `str` `path` correspondiente a la ruta del archivo que contiene la información de las **ordenes**. Retorna un **generador** con instancias de `namedtuple` `Ordenes` asociadas al archivo.

Modificar `def cargar_ordenes_items(path: str) -> Generator:`

Recibe el `str` `path` correspondiente a la ruta del archivo que contiene la información de los **productos** que contiene una **orden**. Retorna un **generador** con instancias de `namedtuple` `OrdenesItems` asociadas al archivo.

Modificar `def cargar_proveedores(path: str) -> Generator:`

Recibe el `str` `path` correspondiente a la ruta del archivo que contiene la información de los **proveedores**. Retorna un **generador** con instancias de `namedtuple` `Proveedor` asociadas al archivo.

Modificar `def cargar_proveedores_productos(path: str) -> Generator:`

Recibe el `str` `path` correspondiente a la ruta del archivo que contiene la información de las relaciones entre los **productos** y los **proveedores**. Retorna un **generador** con instancias de `namedtuple` `ProveedoresProductos` asociadas al archivo.

Esta tarea contendrá *tests* que evaluarán el óptimo de la solución implementada, por lo que estas funciones serán usadas para cargar y crear los generadores que recibirán las consultas definidas en la siguiente sección. Por lo tanto, se recomienda **completar primero estas funciones antes de completar y probar cualquier consulta**.

3.3. Consultas

A continuación se encuentran las consultas que deberás completar en esta tarea haciendo uso de **programación funcional**, separadas por la cantidad de generadores que necesita cada una.

Importante **Casos de empate**

Siempre que haya consultas donde se pueda producir un empate entre los posibles resultados, la **función debe devolver todas las instancias del empate.** Esto aplica para **todas las funciones** donde pueden haber casos de empate.

3.3.1. Consultas Simples

Modificar
```
def productos_desde_fecha(generador_productos: Generator,
    fecha: str, inverso: bool) -> Generator:
```

Recibe un **generador** con instancias de `Productos`, un `str` con una **fecha** de forma `"YYYY-MM-DD"` y el booleano `inverso`.

Retorna un **generador** de productos, los cuales varían según el valor de `inverso`. Si `inverso` es `False` retorna los productos que fueron modificados desde la fecha, incluyéndola, hacia adelante. Mientas que si `inverso` es `True`, retorna los productos modificados desde de la fecha, incluyéndola, hacia atrás.

Modificar
```
def buscar_orden_por_contenido(generador_ordenes_items: Generator,
    id_producto: str, cantidad: int) -> Generator:
```

Recibe un **generador** con instancias de `OrdenesItems`, un `str` que corresponde al **id** de base de datos del producto y un `int` que representa la **cantidad** del producto.

Retorna un **generador** con los `id_base_datos` de las ordenes que contienen el **producto** nombrado y coincide la **cantidad** de producto indicada.

Modificar
```
def proveedores_por_estado(generador_proveedores: Generator,
    estado: str) -> Generator:
```

Recibe un **generador** con instancias de `Proveedor` y un `str` con el **nombre** de un estado.

Retorna un **generador** que contenga los **nombres** de los **proveedores** que están en ese `estado`.

Modificar
```
def ordenes_segun_estado_orden(generador_ordenes: Generator,
    estado_orden: str) -> Generator:
```

Recibe un **generador** con instancias de `Ordenes`, y un `str` con un **estado de orden** en particular.

Retorna un **generador** con todas las ordenes que tengan el **estado de orden** recibido.

Modificar
```
def ordenes_entre_fechas(generador_ordenes: Generator,
    fecha_inicial: str, fecha_final: str) -> Generator:
```

Recibe un **generador** con instancias de `Ordenes`, un `str` que corresponde a la **fecha inicial** a considerar y otro `str` que corresponde a la **fecha final**, ambas en formato `"YYYY-MM-DD"`.

Retorna un **generador** con instancias de `Ordenes` cuyas fechas de creación estén entre las fechas indicadas por `fecha_inicial` y `fecha_final`, ambas inclusive.

En caso de que uno de los parámetros `fecha_inicial` o `fecha_final` sea `"-"`, entonces se debe considerar solamente una de las fechas. Esto quiere decir que, si el parámetro `fecha_final` corresponde a `"-"`, entonces se debe retornar un generador con instancias de `Ordenes` cuyas fechas de creación sean iguales o

posteriores a `fecha_inicial`. Análogamente, si `fecha_inicial` corresponde a `"-"`, se deben considerar las órdenes con fechas de creación iguales o anteriores a `fecha_final`. Se puede asumir que nunca ambas fechas tendrán el valor de `"-"` al mismo tiempo.

Modificar
```
def modificar_estado_orden_ordenes_previas_fecha(generador_ordenes: Generator,
    fecha: str, cambio_estados: dict) -> Generator:
```

Recibe un **generador** con instancias de `Ordenes`, un `str` que representa una **fecha** en formato `"YYYY-MM-DD"`, y un **diccionario** que contiene como llaves un `str` con el **estado actual** de una orden, y como valor un `str` con el **nuevo estado** de la orden. Cada par llave-valor de este diccionario representa una transformación del estado de una orden.

Retorna un **generador** sólo con las instancias de **`Ordenes`** que fueron modificadas. La modificación del estado de la orden debe ocurrir según el cambio que indica el diccionario. Este cambio se debe aplicar **únicamente** a las ordenes cuya fecha sea **anterior** a la fecha recibida.

3.3.2. Consultas Complejas

Modificar
```
def producto_mas_popular(generador_productos: Generator,
    generador_ordenes: Generator, generador_ordenes_items: Generator
    fecha_inicial: str, fecha_final: str, ranking: int) -> Iterable:
```

Recibe tres **generadores**: uno con instancias `Ordenes`, otro con instancias `OrdenesItems` y otro con instancias `Productos`. Luego, recibe dos `str`, uno con **fecha de inicio** y otro con **fecha final**. Por ultimo, recibe un parámetro `int` llamado **ranking** el cual determina el total de elementos a retornar. Este ultimo valor siempre será **mayor a 1**.

Se retorna un iterable que contenga los **nombres** del los **top `ranking` productos** más populares en ese intervalo de tiempo, incluyendo la fecha inicial y final.

El producto mas popular es el que tiene la **mayor cantidad ordenada** a lo largo de todas las `Ordenes` existentes. En este caso se define la cantidad ordenada como la suma de `OrdenesItems.cantidad_productos` de las distintas ordenes. Es decir, no se debe considerar la cantidad de unidades por producto, sino la cantidad de veces que fue ordenado dicho producto. Si se produce un empate en popularidad entre dos productos, se debe realizar un desempate según el atributo `id_base_datos`, ordenándolos de manera descendiente.

Finalmente, si **ranking** es mayor a la cantidad de productos distintos vendidos en el intervalo de fechas, entonces el iterable contendrá menos productos que lo indicado por el *raking*. A si mismo, si en el tiempo indicado no se realizaron ordenes, entonces se retornará un iterable vacío.

Modificar
```
def ordenes_usuario(generador_productos: Generator,
    generador_ordenes: Generator, generador_ordenes_items: Generator,
    ids_usuario: list) -> dict:
```

Recibe tres **generadores**: uno con instancias `Productos`, otro con instancias `Ordenes` y otro con instancias `OrdenesItems`. Ademas, se recibe una lista con **ids de usuarios**, donde estos ids pueden estar repetidos.

Retorna un **diccionario** con los **ids** de los usuarios como llave y como valor otro **diccionario** que contenga el **`id_base_datos`** de un producto como llave y la cantidad total de productos que ordeno el usuario como valor. La cantidad total de productos consiste en la multiplicación de la cantidad de productos pedidos en una orden por la cantidad de unidades que se venden por unidad de medida de un producto en específico.

Modificar `def valor_orden(generador_productos: Generator, generador_ordenes_items: Generator, id_orden: str) -> float:`

Recibe dos **generadores**: uno con instancias de `OrdenesItems` y otro con instancias de `Productos`. Ademas, recibe un `str` con la **id** de una orden.

Retorna un `float` con el **precio total** de la orden.

Modificar `def proveedores_segun_precio_productos(generador_productos: Generator, generador_proveedores: Generator, generador_proveedor_producto: Generator, precio: float) -> list:`

Recibe tres **generadores**: uno con instancias de `Productos`, otro con instancias de `Proveedor` y otro con instancias de `ProveedoresProductos`. Además, recibe un `float` que representa un precio.

Retorna una **lista de tuplas** de la forma **(Proveedor, cantidad-productos)**, donde Proveedor es una instancia de Proveedor, y **cantidad-productos** es un `int` que representa la cantidad de productos ofrecidos por dicho proveedor. Esta lista debe contener a **todos los Proveedores** cuyos productos ofrecidos tengan un precio menor al recibido, es decir, el precio de cada uno de los productos del proveedor debe ser estrictamente menor al precio recibido.

Modificar `def precio_promedio_segun_estado_orden(generador_ordenes: Generator, generador_ordenes_items: Generator, generador_productos: Generator, estado_orden: str) -> float:`

Recibe tres **generadores**: uno con instancias de `Ordenes`, otro con instancias de `OrdenesItems` y otro con instancias de `Productos`. Además, recibe un `str` que representa un **estado de orden**.

Retorna un `float` que represente el **costo total promedio** de las ordenes cuyo estado de orden sea igual al recibido. El promedio final debe estar redondeado al segundo decimal.

Modificar `def cantidad_vendida_productos(generador_productos: Generator, generador_ordenes_items: Generator, ids_productos: list) -> dict:`

Recibe dos **generadores**: uno con instancias de `Productos` y otro con instancias de `OrdenesItems`. Además, recibe una **lista** de `str` que contiene **ids** de productos, los cuales pueden estar repetidos.

Retorna un `dict` cuyas llaves corresponden al **id** de un producto y su valor corresponde a la cantidad de unidades totales vendidas de ese producto según las órdenes dadas en `generador_ordenes_items`. Para determinar la cantidad de unidades totales vendidas de un producto, se debe considerar `cantidad_productos` de los `OrdenesItems` dentro del generador `generador_ordenes_items` y también `cantidad_por_unidad` de las instancias de `Productos` del generador `generador_productos`. Por ejemplo, para un producto con `cantidad_productos` de 10 y `cantidad_por_unidad` de 5, entonces la cantidad de unidades totales vendidas de ese producto será de 50.

Modificar `def ordenes_dirigidas_al_estado(generador_ordenes: Generator, generador_usuarios: Generator, estado: str) -> Generator:`

Recibe dos **generadores**: uno con instancias de `Ordenes` y otro con instancias de `Usuarios`. Además, recibe un `str` correspondiente a un **estado** de Estados Unidos.

Retorna un **generador** con instancias de `Ordenes` pertenecientes a personas que poseen direcciones en el estado entregado. Recuerda que en la sección Subsección 3.1: Datos está la información de cómo se encuentran las direcciones en los datos.

Modificar
```
def ganancias_dadas_por_clientes(generador_productos: Generator,
    generador_ordenes: Generator, generador_ordenes_items: Generator,
    ids_usuarios: list) -> dict:
```

Recibe tres **generadores**: uno primero con instancias de `Productos`, otro con instancias de `Ordenes` y otro con instancias de `OrdenesItems`. Además, recibe una **lista** de `str` que contiene **ids** de usuarios.

Retorna un `dict` cuyas llaves corresponden al **id** de un usuario y cuyos valor corresponde a la **ganancia** generada por todas las órdenes de ese usuario. La ganancia que da un usuario consiste en la suma del dinero gastado a lo largo de todas las órdenes que ese usuario haya realizado.

Modificar
```
def modificar_estados_ordenes_dirigidas_al_estado(generador_ordenes:
    Generator, generador_usuarios: Generator, estado: str,
    cambio_estados_ordenes: dict) -> Generator:
```

Recibe dos **generadores**: uno con instancias de `Ordenes` y otro con instancias de `Usuarios`. Además, recibe un `str` que representa un **estado** de Estados Unidos y un **diccionario** que contiene como llaves un `str` correspondiente a un estado de orden inicial y como valor un `str` con el nuevo estado de dicha orden. Cada par llave-valor en este diccionario representa una transformación del estado de una orden.

Retorna un **generador** con instancias de `Ordenes` correspondientes a usuarios que poseen una dirección dentro del estado de Estados Unidos entregado (`estado`), donde los estados de orden **son modificados** de acuerdo a lo indicado por el diccionario de transición de estados (`cambio_estados_ordenes`).

Modificar
```
def agrupar_items_por_maximo_pedido(generador_productos: Generator,
    generador_ordenes_items: Generator) -> Generator:
```

Recibe dos **generadores**: uno con instancias de `Productos` y otro con instancias de `OrdenesItems`.

Se define **max-cantidad-ordenada** de un producto como el número que representa la cantidad máxima ordenada de un producto entre todas las ordenes que contienen a dicho producto. La **max-cantidad-ordenada** toma en consideración a todas las ordenes donde fue pedido el producto. Este número deberá ser calculado para cada producto. Por ejemplo: si existen tres ordenes distintas que contienen; 1 pan, 2 panes y 3 panes respectivamente, entonces **max-cantidad-ordenada** del producto pan será 3.

Retorna un **generador** que contenga instancias de `Productos`, donde:

- Si la **max-cantidad-ordenada** es igual a 1, entonces no se debe modificar el producto.
- Si la **max-cantidad-ordenada** es mayor a 1, entonces se le deben aplicar las siguientes modificaciones al producto:
  - **Modificar la Unidad de Medida:** Se debe cambiar la unidad de medida según la función `cambio_unidad_medida` disponible en el archivo `utilidades.pyc`, esta función recibe una unidad de medida en `str` y retorna otra unidad de medida en `str`.
  - **Modificar la cantidad por unidad:** Se debe modificar la `cantidad_por_unidad` del producto. La nueva cantidad por unidad pasara a ser la multiplicación entre la **max-cantidad-ordenada** y la cantidad por unidad anterior.
  - **Modificar el Precio:** Se debe modificar el precio del producto para ajustarse al cambio. El nuevo precio del producto sera la **max-cantidad-ordenada** multiplicada por el precio anterior del producto.
  - **Modificar fecha de modificación:** Finalmente, dado que se hicieron modificaciones en el producto, es necesario cambiar el atributo `fecha_modificación` del producto para reflejar la fecha en la que se modifico por última vez el producto. Para obtener la fecha de modificación se

les entrega la función `obtener_fecha_actual` en el archivo `utilidades.pyc`, la cual no recibe *inputs* y se encarga de retornar un `str` con la fecha actual en formato `"YYYY-MM-DD"`.

4. Interfaz Gráfica e Interacción

Con el fin de que el cliente realice las consultas previas sin necesidad de conocimientos técnicos ni de emplear la terminal, se ha pedido diseñar una interfaz gráfica que facilite la interacción y muestre los resultados deseados.

4.1. Modelación del programa

En esta sección se evaluarán, entre otros, los siguientes aspectos:

- Correcta **modularización** del programa, esto quiere decir que se debe respetar y seguir una adecuada estructuración entre ***frontend*** y ***backend***, con un **diseño cohesivo** y de **bajo acoplamiento.**
- Correcto uso de señales entre ***backend*** y ***frontend***.
- Un flujo prolijo a lo largo del programa. Esto quiere decir que el usuario puede navegar sin problemas entre las distintas partes que componen *DCC* solo ejecutando una vez el programa. En otras palabras, un usuario nunca debe quedarse atorado en alguna parte del programa que lo obliga a cerrar *DCC* y volver a ejecutarlo. A modo de ejemplo: si un usuario ingresa a una sección y no se puede mover a otra sección, esto es considerado como una mala implementación.
- Una interfaz interactiva integrada con las funcionalidades pedidas. Esto quiere decir que se espera que la comunicación y el comportamiento del programa sea visible y controlable desde la interfaz. **No se asignará puntaje** si las funcionalidades solicitadas no son visibles en la interfaz ó si es que estas solo se pueden comprobar en la consola o en el código, a menos que se indique explícitamente lo contrario.
- Generación de las ventanas mediante código programado por el estudiante. Es decir, **no se permite** la creación de ventanas con el apoyo de herramientas como `QtDesigner`, `QML`, entre otros.

4.2. Ventanas

Para la correcta implementación de *DCC*, se espera la creación de **dos ventanas**, cada una con elementos mínimos de interfaz que deben estar presentes, los cuales serán detallados a continuación.

Importante

Los esquemas y diseños que serán expuestos son únicamente referenciales. No es necesario que tu tarea sea una copia exacta de estos; por lo que **lo único que será evaluado es que los elementos mínimos estén presentes y funcionen del modo que se exige**.

Los elementos mínimos de cada ventana serán explicitados en la sección correspondiente; pero en ningún caso se les exige cosas relacionadas a la estética de la interfaz gráfica. Por lo tanto, una ventana que sólo tenga los elementos expuestos en los esquemas, tiene el mismo nivel de validez que otra llena de decoraciones y efectos interactivos, esto, suponiendo que ambas tengan el comportamiento deseado.

En caso de que lo consideres necesario puedes ubicar los elementos en otras posiciones; agregar creatividad inventando botones nuevos; nuevas interacciones, diseños y hasta animaciones. Cabe destacar que si se desea desarrollar funcionalidades extras, estas serán evaluadas bajo los mismos criterios generales mencionados anteriormente. Esto quiere decir que, si el programa falla debido a una funcionalidad extra, se aplicara el descuento en el *item* correspondiente.

4.2.1. Ventana de Entrada

Esta ventana existe antes de llegar a la ventana principal, y se muestra al iniciar el programa. Se muestra un mensaje de bienvenida, con un botón para pasar a la ventana siguiente.

En la Figura 1a se muestra un ejemplo de cómo se puede estructurar esta ventana.

4.2.2. Ventana Principal

Esta ventana sirve para elegir de manera interactiva las consultas del ***backend*** que serán ejecutadas y, mostrar los resultados de dichas consultas.

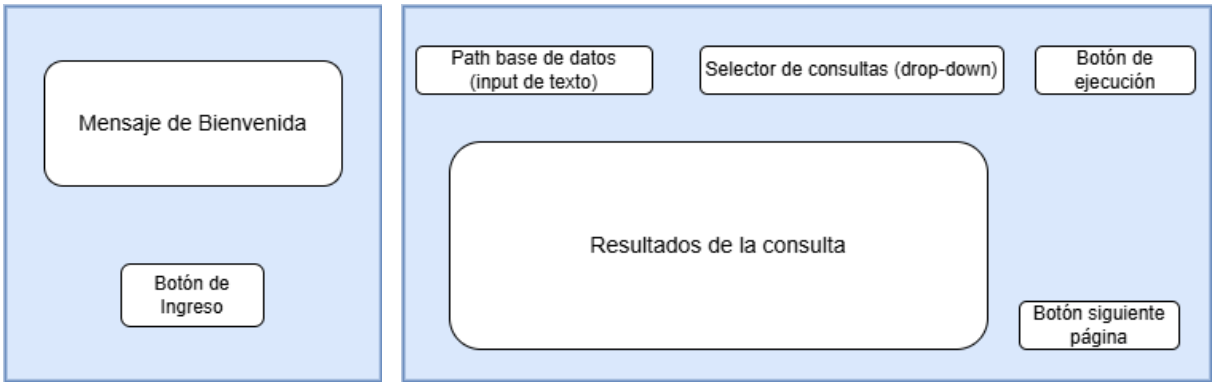
Dado que la gran mayoría de las consultas retornan un iterable, el formato de presentación de los resultados consistirá en mostrar un texto, donde cada línea corresponda a un elemento del iterable.

También se debe implementar un sistema de paginación, para así manejar el caso de que los resultados sean muy extensos. Esto se debe hacer mostrando una cantidad fija de elementos por página, junto a un botón que permite pasar a la página siguiente, es decir, cargar el siguiente conjunto de elementos. Dado que alguno de los resultados son generadores y, que estos solo pueden ser recorridos una vez, no es necesario implementar un botón para regresar a la página anterior.

Para esto, se debe crear una ventana que contenga como mínimo los siguientes elementos (los nombres son referenciales, puedes elegir el que quieras):

- ***Input* de texto** que recibe el *path* relativo al directorio con los datos a utilizar en las consultas.
- ***Drop-down*** con la lista de consultas disponibles. Permite seleccionar una consulta a ejecutar.
- **Botón "Ejecutar Consulta"** que se encarga de cargar los datos indicados en el *input* de texto y ejecutar la consulta seleccionada en el *drop-down*.
- **Área de texto** que posea -por lo menos- un *scroll* de tipo vertical. Este elemento debe permitir ver la cantidad de datos -que corresponden al resultado de la consulta ejecutada- acordes a la página actual. En caso de que no se puedan apreciar todos los datos dentro del elemento, se debe poder acceder a ella por medio del *scroll*.
- **Botón "Siguiente Página"** para avanzar en la paginación.

En la Figura 1b se muestra un ejemplo de cómo puede verse la ventana principal, con los elementos previamente mencionados.



(a) Ventana de entrada (b) Ventana principal

Figura 1: *Mock-up* de las ventanas

5. *Tests*

El objetivo principal de la implementación de los *tests* es la eficiencia. Cada test posee un tiempo límite de ejecución permitido. Estos tiempos varían desde 2 segundos en los *tests* más fáciles hasta 22 en los mas complicados. **Si no se cumple con el tiempo límite, no se otorgará puntaje.**

Los *tests* presentes en esta evaluación se dividirán en dos conjuntos:

- **Correctitud:** Evaluará el comportamiento de la solución implementada con respecto a posibles casos bordes. En estos casos, se probará la función con generadores ya cargados por el programa.
- **Carga de datos:** Evaluará el comportamiento de la solución implementada con respecto a la eficiencia del código y su capacidad de trabajar con archivos de diversos tamaños.

Para lograr este objetivo de esta evaluación, es esencial un correcto uso de **generadores** y **programación funcional**. El no aplicar correctamente los contenidos anteriormente mencionados, puede provocar que las funciones no terminen en el tiempo esperado.

Reiterando lo indicado en el Flujo del programa: Es sumamente importante que **no realices la tarea basándote en los tests públicos**. Te debes basar en el enunciado y sólo puedes usar los tests públicos para apoyo. Debido a esto, es importante destacar que **si el alumno tiene los test públicos correctos, pero los tests privados incorrectos, no se otorgará puntaje**.

5.1. Ejecución de *tests*

Para la corrección automática se entregarán varios archivos `.py` los cuales contienen diferentes *tests* que ayudan a validar el desarrollo de la tarea. Para ejecutar estos *tests*, primero debes posicionar tu terminal/consola en la carpeta de la tarea `Tareas/T3/`. Luego, desde esta misma, debes escribir el siguiente comando para ejecutar todos los *tests*:

- `python3 -m unittest discover tests_publicos -v -b`

En cambio, si deseas ejecutar un subconjunto de *tests*, puedes hacerlo escribiendo lo siguiente:

- `python3 -m unittest -v -b tests_publicos.<test_N>`
  Reemplazando `<test_N>` por el test que desees probar.

Por ejemplo, si quisieras probar si realizaste correctamente la función `producto_mas_popular`, deberás escribir lo siguiente:

- `python3 -m unittest -v -b tests_publicos.test_07_producto_mas_popular_correctitud`.

Importante: Recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `python`, `py3` o `python3.11`.

6. *utilidades.pyc*

Se entregara un archivo **.pyc**, el cual contendrá las `namedtuples` y funciones que deben utilizar para el desarrollo de la tarea. Estas `namedtuples` y funciones solo deben ser importadas y utilizadas, sin la posibilidad de ser modificadas o ver su funcionamiento.

6.1. `Namedtuples`

Tal como se indica en la Subsección 3.1: Datos, cada uno de los datos entregados en los archivos CSV presenta una `namedtuples` que lo representa. El detalle de estas `namedtuples` se encuentra en la Subsección 3.1.

6.2. Funciones Auxiliares

Adicionalmente, para que puedas implementar algunas consultas se te entregan las siguientes funciones auxiliares:

- No modificar `def fecha_actual() -> str:`

  No recibe parámetros y retorna la fecha actual en un formato válido con la tarea.

- No modificar `def cambio_unidad_medida(unidad_medida_actual: str) -> str:`

  Recibe un `str` correspondiente a una unidad de medida válida de un producto y retorna una nueva unidad de medida válida en base a la entregada.

7. `.gitignore`

Para esta tarea **deberás utilizar un** `.gitignore` para ignorar los archivos indicados, este deberá estar dentro de tu carpeta `Tareas/T3/`.

Los elementos que no debes subir y **debes ignorar mediante el archivo** `.gitignore` para esta tarea son:

- El enunciado.
- La carpeta `data/` y los archivos `csv` y `zip` correspondientes a los datos.
- La carpeta `test_publicos/` y los archivos zip relacionados a dicha carpeta.
- Cualquier archivo `utilidades.pyc` o algún otro que posea dicha extensión.

Recuerda **no ignorar archivos vitales de tu tarea como los que tú creas o modificas, o tu tarea no podrá ser revisada**.

El correcto uso del archivo `.gitignore`, implica que los archivos **deben** no subirse al repositorio debido al uso archivo `.gitignore` y no debido a otros medios.

Importante **Debes asegurarte de que ni los archivos de datos ni los archivos de los *tests* no sean subidos a tu repositorio personal, en caso contrario, se aplicarán 10 décimas de descuento de formato en la corrección de la evaluación.** Dado que en esta evaluación presenta archivos de gran tamaño, junto a los archivos base de la tarea se incluye un archivo `.gitignore` que ignora todos los archivos `csv` y `zip`.

Finalmente, en caso hacer *commit* de la carpeta "`data/`" o "`tests_publicos/`", debido al tamaño de dichos archivos, Git impedirá que dicho *commit* y los siguientes puedan ser subidos al repositorio remoto,

por lo que no podrán hacer entregas parciales. En caso de que lleguen a enfrentarse a este problema, deben:

1. Hacer un respaldo de su solución.
2. Volver a clonar su repositorio personal.
3. Agregar al nuevo repositorio los cambios respaldados.
4. Hacer *commit* y *push* de los cambios, teniendo consideración de no agregar los archivos de datos.

8. Importante: Corrección de la tarea

En el **siguiente enlace** se encuentra la distribución de puntajes. Para esta tarea, el carácter funcional del programa será el pilar de la corrección, es decir, **sólo se corrigen tareas que puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y *push* en sus repositorios, corroborando que cada *test* que les pasamos para cada consulta corra en el tiempo indicado (Sección 5: *Tests*), en caso contrario se asumirá un resultado incorrecto.

Importante: Todo ítem corregido automáticamente será evaluado de forma ternaria: puntaje completo si pasa todos los *tests* de dicho ítem, medio punto para quienes pasan más del 65 % de los *test* de dicho ítem, y 0 puntos para quienes no superan el 65 % de los *tests* en dicho ítem. Finalmente, todos los descuentos serán asignados automáticamente por el cuerpo docente.

La corrección se realizará en función del último *commit* realizado antes de la fecha oficial de entrega (viernes 30 de mayo a las 20:00). Si se desea continuar con la evaluación en el periodo de entrega atrasado, es decir, realizar un nuevo *commit* después de la fecha de entrega, **es imperante responder el formulario de entrega atrasada** sin importar si se utilizará o no cupones. Responder este formulario es el mecanismo que el curso dispone para identificar las entregas atrasadas. El enlace al formulario está en la primera hoja de este enunciado y estará disponible para responder hasta el domingo 1 de junio las 23:59.

Para terminar, si durante la realización de tu tarea se te presenta algún problema o situación que pueda afectar tu rendimiento, no dudes en contactar al ayudante de Bienestar de tu sección. El correo está en el **siguiente enlace**.

9. Restricciones y alcances

- Esta tarea es **estrictamente individual**, y está regida por el **Código de honor de Ingeniería**.
- Tu programa debe ser desarrollado en Python 3.11.X con X mayor o igual a 7.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py` que estén correctamente ordenados por carpeta. **No se revisará archivos en otra extensión como `.ipynb`**.
- Todo el código entregado debe estar contenido en la carpeta y rama (*branch*) indicadas al inicio del enunciado. Ante cualquier problema relacionado a esto, es decir, una carpeta distinta a `Tareas/T3/` o una rama distinta a `main`, se recomienda preguntar en las ***issues* del foro**.
- Si no se encuentra especificado en el enunciado, supón que el uso de cualquier librería Python está prohibida. Pregunta en la *issue* especial del **foro** si es que es posible utilizar alguna librería en particular.
- Debes adjuntar un **único archivo markdown**, llamado `README.md`, **conciso y claro**, donde describas las referencias a código externo. El no incluir este archivo, incluir un readme vacío o el subir más de un archivo `.md`, conllevará un **descuento** en tu nota.
- Esta tarea se debe desarrollar **exclusivamente** con los contenidos liberados al momento de publicar el enunciado. No se permitirá utilizar contenidos que se vean posterior a la publicación de esta evaluación.
- Se encuentra estrictamente prohibido citar código que haya sido publicado **después de la liberación del enunciado**. En otras palabras, solo se permite citar contenido que ya exista previo a la publicación del enunciado. Además, se encuentra estrictamente prohibido el uso de herramientas generadoras de código para el apoyo de la evaluación.
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro que sí sea especificado por enunciado.

Las tareas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (1,0)**.**