



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

IIC2233 Programación Avanzada (2024-1)

Tarea 3

Entrega

- Tarea y README.md
 - **Fecha y hora oficial (sin atraso):** martes 15 de octubre de 2024, 20:00.
 - **Fecha y hora máxima (2 días de atraso):** jueves 17 de octubre de 2024, 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](#).
- **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`.

Índice

1. <i>Little DCCaesars</i>	3
2. Flujo del programa	3
3. Programación Funcional	5
3.1. Datos	5
3.1.1. Pizzas	5
3.1.2. Locales	5
3.1.3. Pedidos	5
3.1.4. ContenidoPedidos	6
3.2. Carga de datos	6
3.3. Consultas	7
3.3.1. Consultas que reciben un generador	7
3.3.2. Consultas que reciben dos generadores	9
3.3.3. Consultas que reciben tres o más generadores	10
3.3.4. Consultas anidadas	12
4. <i>Tests</i>	13
4.1. Ejecución de <i>tests</i>	13
5. <i>.gitignore</i>	14
6. README	14
7. Importante: Corrección de la tarea	15
8. Restricciones y alcances	15



1. *Little DCCaesars*

Los zombies de “Plants vs. Zombies”, después de años dedicados a devorar plantas y humanos, finalmente se aburrían de esa monótona rutina y decidieron embarcarse en un nuevo negocio: abrir un restaurante de pizza llamado *Little DCCaesars*. Con gran entusiasmo, decoraron el local a su estilo, con mesas hechas de viejas lápidas y un horno a leña que en realidad era una antigua fogata utilizada en noches de ataque. Ofrecían todo tipo de sabores únicos, como la “Pizza de cerebro con champiñones” y la “Especial de girasol con salsa secreta”.

Al principio, las cosas parecían ir bien. Los zombies cocinaban con dedicación y los clientes, aunque algo asustados, no podían negar que las pizzas eran deliciosas. Sin embargo, muy pronto se dieron cuenta de que, aunque hacer pizza era divertido, llevar las cuentas y la administración del negocio no era su fuerte. Las facturas empezaron a acumularse en pilas desordenadas, no lograban analizar datos provenientes de su gran cantidad de franquicias, y los pedidos siempre llegaban tarde o completamente equivocados. Más de una vez, un cliente que había pedido una “Pizza margarita” terminaba recibiendo una “Pizza de sesos”, cortesía de un distraído zombie cocinero.

Desesperados por la inminente quiebra y el caos organizativo, los zombies decidieron pedir ayuda. Te contactaron, sabiendo que eras la única persona capaz de poner orden en sus finanzas y salvar el restaurante. Ahora te enfrentas a una tarea monumental: organizar sus cuentas y obtener información valiosa para el negocio zombie, todo mientras intentas evitar que los zombies te vean como su próxima comida. ¡Es hora de salvar el negocio de pizza de los zombies antes de que se hunda por completo!

2. Flujo del programa

En *Little DCCaesars*, deberás acceder a datos sobre las pizzas y los pedidos, los cuales están almacenados en archivos de distintos tamaños. Luego deberás completar, utilizando **correctamente programación funcional y generadores**, una serie de consultas que permitirán obtener distintos tipos de información sobre las distintas franquicias del local.

Para ~~facilitar la corrección~~ lograr un código más ordenado, se pedirá que implementes como **mínimo** ciertas funciones. Estas funciones ya están definidas en los archivos que te entregamos para esta evaluación. Solo debes completar estas funciones, es decir, **no debes cambiar su nombre, alterar los argumentos recibidos o cambiar lo que retornen**. Puedes crear nuevos archivos y/o funciones si estimas conveniente. También se permite crear funciones en otro módulo y que las funciones que pedimos completar únicamente llame a esa función externa. El requisito primordial es que debes mantener el formato de las funciones informadas en este enunciado, asegurando que estas puedan ejecutarse según lo indicado en la [Subsección 4.1: Ejecución de tests](#).

Además, la corrección de esta tarea **será únicamente mediante el uso de tests**, los cuales otorgarán puntaje dependiendo de cuántos tests se pasen y el tipo de estos, para cada una de las funcionalidades a implementar. Para apoyar el desarrollo de esta tarea, se provee de una batería de tests donde podrán revisar distintos casos con su respuesta esperada. Estos tests públicos corresponden a un segundo chequeo de la evaluación, es decir, no son representativos de todos los casos posibles que tiene una función. Por lo tanto, **es responsabilidad del estudiantado confeccionar una solución que cumplan con lo expuesto en el enunciado y que no esté creada solamente a partir de los tests públicos**. De ser necesario, el estudiantado deberá pensar en nuevos casos que sean distintos a los tests públicos. **No se aceptarán supuestos que funcionaron en los tests públicos, pero van en contra de lo expuesto en el enunciado.**

En el directorio de la tarea encontrarás los siguientes archivos y directorios:

- **Modificar** `consultas.py`: Este archivo contiene las funciones a completar señaladas en la sección [Programación Funcional](#).
- **No modificar** `data/`: Esta carpeta contendrá una serie de archivos `csv` dentro de otras carpetas. Cada subcarpeta tiene archivos de tamaño distinto, estos archivos son los necesarios para realizar las consultas. Estos archivos se deben abrir usando el *encoding utf-8*.

En un comienzo, esta carpeta estará vacía. Debes descargar los archivos desde [este enlace](#). Se encontrarán en formato `zip` y debes descomprimir la carpeta asegurando que el contenido de los archivos presente la siguiente estructura:

```
T3/
├── data/
│   ├── l/
│   │   ├── pizzas.csv
│   │   ├── locales.csv
│   │   └── ...
│   ├── m/
│   │   ├── pizzas.csv
│   │   ├── locales.csv
│   │   └── ...
│   ├── s/
│   └── xl/
├── tests_publicos/
├── .gitignore
├── consultas.py
└── README.md
```

- **No modificar** `test_publicos/`: Este directorio contiene los distintos *tests* de la evaluación. Hay dos tipos de *tests*, de correctitud y de manejar datos.
- **No modificar** `utilidades.py`: Este archivo contiene la implementación de las *namedtuples* a utilizar.

3. Programación Funcional

Para poder analizar a la cadena de restaurantes de *Little DCCaesars*, sus clientes y sus ventas, se necesitará de tu ayuda experta para obtener información mediante la realización de diversas consultas, aplicando tus conocimientos de programación funcional.

Para esto, deberás interactuar con distintos tipos de datos, los que serán explicados con mayor detalle en [Datos](#) y completar distintas funciones pedidas en [Consultas](#), que se realizará mediante la modificación de un código pre-existente.

3.1. Datos

Para que puedas implementar las funcionalidades, tendrás que interactuar con las siguientes *namedtuples* que se encuentran presentes en el archivo `utilidades.py`:

3.1.1. Pizzas

Indica la información de una pizza. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
nombre	<code>str</code>	Nombre de la pizza junto al tamaño de esta. Estos dos aspectos estarán separados por un guión bajo (<code>_</code>) y será el único guión bajo del nombre.	"Pepperoni Clásica_S"
ingredientes	<code>str</code>	Ingredientes que contiene la pizza, dividido con <code>"; "</code> .	"pepperoni;queso;salsa de tomate"
precio	<code>int</code>	Precio de la pizza.	12239

3.1.2. Locales

Indica la información de los locales. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
id_local	<code>int</code>	Identificador único del local.	334
dirección	<code>str</code>	Dirección del local.	"0176 Petty Islands"
pais	<code>str</code>	País donde se ubica el local.	"Algeria"
ciudad	<code>str</code>	Ciudad del local.	"New Rhondaview"
cantidad_trabajadores	<code>int</code>	Cantidad de trabajadores en el local.	13

3.1.3. Pedidos

Indica la información de cada pedido. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
id_pedido	<code>int</code>	Identificador único del pedido.	21
id_local	<code>int</code>	Identificador único del local.	132
id_cliente	<code>int</code>	Identificador único de un cliente.	33
fecha	<code>str</code>	Fecha del pedido, en el formato YYYY-MM-DD.	"2024-02-19"
hora	<code>str</code>	Hora del pedido, en el formato HH:MM:SS.	"06:33:45"

3.1.4. ContenidoPedidos

Indica el contenido de cada pedido. Posee los atributos:

Atributo	Tipo	Descripción	Ejemplo
id_pedido	int	Identificador único del pedido.	18
nombre	str	Nombre de la pizza pedida.	"Vegetariana_M"
cantidad	int	Cantidad de pizzas del tipo en el pedido.	1
descuento	float	Descuento incluido en el pedido.	0.13

3.2. Carga de datos

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

En la carpeta **data** que está dentro de **test_publicos**, podrás encontrar cuatro subcarpetas **s**, **m**, **l** y **xl** con bases de datos de distintos tamaños: pequeños, medianos, grandes y muy grandes respectivamente. Deberás copiar esta carpeta y ubicarla en tu carpeta de la tarea T3.

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

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

Recibe el *path* al archivo que contiene la información de las pizzas. Retorna un generador con instancias de *namedtuple* `Pizza` asociadas al archivo.

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

Recibe el *path* al archivo que contiene la información de los locales. Retorna un generador con instancias de *namedtuple* `Local` asociados al archivo.

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

Recibe el *path* al archivo que contiene la información de los pedidos. Retorna un generador con instancias de *namedtuple* `Pedido` asociados al archivo.

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

Recibe el *path* al archivo que contiene la información del contenido de cada pedido. Retorna un generador con instancias de *namedtuple* `ContenidoPedido` asociadas al archivo.

Esta tarea contendrá *tests* que evaluarán el óptimo de la solución implementada, por lo que estas funciones serán usada 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 poder 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 hayan consultas donde se pueda producir un empate entre los posibles resultados, la **función debe devolver todas las instancias del empate**. Por ejemplo, en la función `pedido_con_mayor_descuento_utilizado`, deberá retornar todos los pedidos con mayor descuento. Esto aplica para **todas las funciones** donde pueden haber casos de empate.

3.3.1. Consultas que reciben un generador

```
Modificar def pedidos_con_al_menos_esta_pizza(generator_contenido_pedidos: Generator,
                                              tipo_de_pizza: str) -> Iterable:
```

Recibe un **generador** con instancias de `ContenidoPedido` y un `str` que corresponde al **tipo** de pizza entregada. Este último parámetro (`tipo_de_pizza`) consiste en el nombre de la pizza pero sin considerar su tamaño (e.g. Pepperoni Clásica, Vegetariana, entre otros).

Retorna un **iterable** con instancias de `ContenidoPedido` que posean al menos una pizza del tipo `tipo_de_pizza`. En caso de que no haya ningún pedido que posea una pizza con el nombre entregado se debe retornar un **iterable** vacío.

```
Modificar def cantidad_vendida_de_pizza_por_tipo(
    generator_contenido_pedidos: Generator, tipo_de_pizza: str) -> int:
```

Recibe un **generador** con instancias de `ContenidoPedido` y un `str` que corresponde al tipo de una pizza. Este último parámetro, `tipo_de_pizza`, corresponde al nombre de la pizza pero sin considerar su tamaño (e.g. Pepperoni Clásica, Vegetariana, entre otros).

Retorna un `int` con la cantidad total vendida de pizza del tipo `tipo_de_pizza` a partir del **generador** entregado.

```
Modificar def pedido_con_mayor_descuento_utilizado(
    generator_contenido_pedidos: Generator) -> Iterable:
```

Recibe un **generador** con instancias de `ContenidoPedido`.

Retorna un **iterable** con instancias de `ContenidoPedido` que poseen el mayor descuento utilizado dentro de los pedidos.

```
Modificar def ajustarPrecioSegunIngredientes(generator_pizzas: Generator,
                                              ingrediente: str, diferenciaPrecio: int) -> Iterable:
```

Recibe un **generador** con instancias de `Pizza`, el **nombre** de un ingrediente y un **valor** que representa en cuánto debe cambiar el precio de la pizza. Este valor puede ser positivo (para aumentar el precio de la pizza) o negativo (para reducir su precio). El precio de una pizza nunca puede estar por debajo de \$7000.

Retorna un **iterable** con las instancias de `Pizza` que tengan ese ingrediente con el precio modificado.

Hint: Recuerda que las *namedtuples* son inmutables, por lo que no se podrá modificar el valor directamente.

Modificar `def clientes_despues_hora(generator_pedidos: Generator, hora: str) -> str:`

Recibe un **generador** con instancias de `Pedido` y un `str` con el valor de la hora de la forma `"HH"`.

Retorna un `str` con el id de los clientes que hayan hecho un pedido entre el inicio de la hora dada y el fin del día, separados por un espacio.

Por ejemplo si solo clientes 1, 15 y 49 cumplen, se retorna `"1 15 49"`. El orden de los ids no importa.

Modificar `def cliente_indeciso(generator_pizzas: Generator, ingrediente_no_deseado: str, cantidad_pizzas: int) -> Iterable:`

Recibe un **generador** de instancias de `Pizza`, un `str` con el nombre de un ingrediente que el cliente no quiere en sus pizzas y un `int` con la cantidad de pizzas que quiere el cliente. Debe elegir las pizzas una a una según el orden de `generator_pizzas`. si no se llegó a `cantidad_pizzas` y se llegó al final del generador, debe continuar desde el principio del mismo.

Retorna un **iterable** con `cantidad_pizzas` instancias de `Pizza` que no tengan `ingrediente_no_deseado`. Se debe incluir las instancias repetidas y en caso de que no hayan pizzas con tal ingrediente, retorna un iterable vacío.

Modificar `def pizzas_con_ingrediente(generator_pizzas: Generator, ingrediente: str) -> Iterable:`

Recibe un **generador** con instancias de `Pizza` y un `str` con el nombre de un ingrediente.

Retorna un **iterable** con las instancias de `Pizza` que contengan el ingrediente recibido.

Modificar `def pizzas_pagables_de_un_tamano(generator_pizzas: Generator, dinero: int, tamano: str) -> Iterable:`

Recibe un **generador** con instancias de `Pizza`, un `int` que indica una cantidad de dinero y un `str` que corresponde al tamaño de pizza.

Retorna un **iterable** con las instancias de `Pizza` del tamaño pedido que cuyo precio no supere el dinero disponible.

Modificar `def cantidad_empleados_pais(generator_locales: Generator, pais: str) -> int:`

Recibe un **generador** con instancias de locales, y un `str` con el nombre de un país.

Retorna un `int` con la cantidad total de empleados de *Little DCCaesars* en el país pedido, o sea, la suma de los empleados de cada local ubicado en ese país.

3.3.2. Consultas que reciben dos generadores

```
Modificar def ganancias_producidas_en_los_pedidos(  
    generador_contenido_pedidos: Generator,  
    generador_pizzas: Generator) -> Iterable:
```

Recibe dos **generadores**, el primero con instancias de `ContenidoPedido` y el segundo con instancias de `Pizza`.

Retorna un **iterable** con tuplas (`tuple`), en donde el primer elemento de cada una de estas tuplas corresponde al **id** de un pedido y el segundo elemento corresponde a la ganancia que se genera con ese pedido.

La ganancia de cada pedido estará determinada por: la cantidad pedida de una determinada pizza, el tipo y tamaño de dicha pizza y el eventual descuento utilizado. En caso de que alguna ganancia sea un número decimal (`float`) se debe aproximar por redondeo con la función *built-in* `round` a la cifra de la unidad para que se obtenga una ganancia entera (`int`).

```
Modificar def pizza_mas_vendida_del_dia(generador_contenido_pedidos: Generator,  
    generador_pedidos: Generator, fecha: str) -> Iterable:
```

Recibe dos **generadores**, uno con instancias de `ContenidoPedido` y otro con instancias de `Pedido`, y un `str` que corresponde a una **fecha** en formato `"YYYY-MM-DD"`.

Retorna un **iterable** con el **tipo de pizza** (`str`) que corresponde al nombre de la pizza (sin considerar su tamaño) que tuvo más unidades vendidas en el día entregado en `fecha`. En caso de que no haya habido ventas o pedidos realizados en la fecha entregada se debe retornar un iterable vacío.

```
Modificar def pizza_del_mes(generador_pedidos: Generator,  
    generador_contenido_pedidos: Generator, mes: str) -> Iterable:
```

Recibe dos **generadores**, con instancias de `Pedido` y `ContenidoPedido`, y un `str` `mes` de la forma `"MM"`.

Retorna un **iterable** de el nombre del tipo de pizza más vendida durante el mes, es decir, sin considerar su tamaño. En caso de que no haya habido ventas o pedidos realizados en el mes indicado, se debe retornar un iterable vacío.

```
Modificar def popularidad_mezcla_de_ingredientes(generador_pizzas: Generator,  
    generador_contenido_pedidos: Generator, ingredientes: set) -> int:
```

Recibe dos **generadores**, con instancias de `Pizza` y `ContenidoPedido`, y un **conjunto** de nombres de ingredientes.

Retorna un `int` que corresponde a la cantidad de pizzas vendidas que tenían al menos, todos los ingredientes del `set`.

```
Modificar def total_ahorrado_pedidos(generador_contenido_pedidos: Generator,  
    generador_pizzas: Generator) -> int:
```

Recibe un **generador** con instancias de `contenido pedidos` y un **generador** con instancias de `pizzas`.

Retorna un `int` que represente la cantidad de dinero ahorrada a través de los descuentos aplicados en cada pedido. Por ejemplo, si un pedido tiene 2 pizzas que cuestan \$10.000, y un descuento de 0.2, la cantidad de dinero ahorrada sería \$4.000. En caso de que lo ahorrado corresponda a un número decimal (`float`) se debe aproximar por redondeo con la función *built-in* `round`.

```
Modificar def pizza_favorita_cliente(generator_pedidos: Generator,
                                     generator_contenido_pedidos: Generator, id_cliente: int) -> Iterable:
```

Recibe un **generador** de Pedido, un **generador** con instancias de ContenidoPedido y un **int** con el id de un cliente.

Retorna un **iterable** con tuplas de la forma (**nombre**, **cantidad**), donde nombre es un **str** con el nombre de la pizza más comprada por el cliente, y cantidad es un **int** con la cantidad de veces que esta pizza fue pedida. En caso de que no haya una pizza favorita, se retorna un iterable vacío.

Como consideración, el nombre de la pizza es independiente de su tamaño. Por ejemplo, si un cliente sólo pidió 2 Vegetarianas M y 1 Vegetariana S, la tupla retornada deberá ser (Vegetariana, 3).

3.3.3. Consultas que reciben tres o más generadores

```
Modificar def local_mas_pizzas_vendidas_por_tipo_de_pizza(
    generator_contenido_pedidos: Generator, generator_pedidos: Generator,
    generator_locales: Generator, tipo_de_pizza: str) -> Iterable:
```

Recibe tres **generadores**: el primero de ellos posee instancias de ContenidoPedido, el segundo instancias de Pedido y el tercero instancias de Local. Además, recibe un **str** que corresponde al tipo de la pizza sin considerar su tamaño.

Retorna un **iterable** con **instancias** de Local que correspondan a los locales que hayan tenido, en total, el mayor número de unidades de pizza vendidas del tipo **tipo_de_pizza** en los pedidos. En caso de que no haya pedidos en los que se tengan pizzas del tipo **tipo_de_pizza** se debe retornar un iterable vacío.

```
Modificar def ganancia_total_de_un_local(generator_contenido_pedidos: Generator,
    generator_pedidos: Generator, generator_pizzas: Generator,
    id_local: int) -> int:
```

Recibe tres **generadores**: el primero de ellos contiene instancias de ContenidoPedido, el segundo posee instancias de Pedido y el tercero instancias de Pizza. Además, recibe el **id** de un local.

Retorna un **int** que corresponde a la ganancia total que tuvo el local correspondiente al **id** entregado, dados los pedidos dentro del **generador** de pedidos. En caso de que la ganancia del local resulte un número decimal (**float**), se debe aproximar por redondeo con la función *built-in* **round** el resultado a la cifra de la unidad para que se tenga una ganancia entera (**int**). En caso de que el local no haya tenido ventas se debe retornar 0.

```
Modificar def promedio_ventas_con_descuento_de_un_pais(
    generator_contenido_pedidos: Generator, generator_pedidos: Generator,
    generator_locales: Generator pais: str) -> float:
```

Recibe tres **generadores** con instancias de ContenidoPedido, Pedido y Local, y un **str** con el nombre de un país.

Retorna un **float** redondeado a dos decimales con **round**, que corresponde al promedio de los descuentos de todos los pedidos del país especificado. Al momento de calcular el promedio, solo debe considerarse el descuento una vez por pedido.

```
Modificar def gasto_cliente_por_mes(generator_contenido_pedidos: Generator,  
                                     generator_pedidos: Generator, generator_pizzas: Generator  
                                     id_cliente: int, year: int) -> list:
```

Recibe tres **generadores** con instancias de ContenidoPedido, Pedido y Pizza, y dos **int** que contienen un id de un cliente y un año.

Retorna una **lista** que contiene la cantidad de dinero que gastó el cliente durante ese año, separado por mes, es decir, una lista con 12 valores, uno para cada mes del año. Si el cliente no compró pizza durante algún mes del año, el gasto de dicho mes será \$0. Los meses deben estar ordenados de forma ascendente partiendo desde enero.

```
Modificar def pizzas_vendidas_mes_pais(generator_contenido_pedidos: Generator,  
                                         generator_pedidos: Generator, generator_locales: Generator  
                                         pais: str, mes: int, year: int) -> int:
```

Recibe tres **generadores** con instancias de ContenidoPedido, Pedido y Local, un **str** correspondiente al nombre de un país y dos **int** que representan el número de un mes y un año.

Retorna un **int** que corresponde a la cantidad total de pizzas que se vendieron en el país durante el mes y año indicados.

3.3.4. Consultas anidadas

Modificar `def consulta_anidada(instrucciones: dict) -> Any:`

Recibe un `dict` con dos o más llaves, donde estas representarán una función y los argumentos que esta debe recibir. Las funciones contenidas en este diccionario de instrucciones corresponderán a: cualquiera de las funciones de [Carga de datos](#) y las siguientes funciones de [Consultas](#):

- `cliente_indeciso`
- `pizzas_con_ingrediente`
- `pizzas_pagables_de_un_tamano`
- `cantidad_empleados_pais`
- `total_ahorrado_pedidos`

Como se dice anteriormente, dentro de este `dict` estarán los argumentos necesarios para poder ejecutar una función, los cuales pueden corresponder a valores (`str`, `int`, etc) o un nuevo diccionario de **instrucciones**. Este último caso corresponde a consultas anidadas.

Retorna el resultado de aplicar las consultas anidadas. Este resultado puede ser un generador, un valor o una instancia de *namedtuple*.

Por ejemplo, a partir del siguiente diccionario de instrucciones:

```
1 {  
2     funcion: 'pizzas_pagables_de_un_tamano',  
3     generador_pizzas: {  
4         funcion: 'pizzas_con_ingredientes',  
5         generador_pizzas: {  
6             funcion: 'cargar_pizzas'  
7             path: 'data/l/pizzas.csv'  
8         },  
9         ingrediente: 'tomate'  
10    },  
11    dinero: 10000,  
12    tamaño: 'S'  
13 }
```

Entonces se deberá cargar un generador de pizzas a partir del *path* indicado. Luego, la función `pizzas_con_ingrediente` filtrará todas las pizzas que contengan tomate. Finalmente, se aplica la función `pizzas_pagables_de_un_tamaño` y retorna un generador con todas las pizzas de tamaño S que pueden ser compradas con \$10.000.

4. *Tests*

El objetivo principal de la implementación de los *tests* es eficiencia. Se evaluará que cada *test* corra en menos de 30 segundos. **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.

4.1. 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/console 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 `pizza_del_mes`, deberás escribir lo siguiente:

- `python3 -m unittest -v -b tests_publicos.test_12_pizza_del_mes_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`.

5. .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`
- La carpeta `test_publicos/`
- El archivo `utilidades.py`

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`.

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 los archivos de datos no sean subidos a tu repositorio personal, en caso contrario, se aplicarán 10 décimas de descuento de formato en la corrección de la evaluación.

Finalmente, en caso hacer *commit* de la carpeta “`data/`”, debido a los archivos de datos, 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.

6. README

Debido al caracter automatizado de la corrección de esta tarea, los archivos README no serán revisados manualmente por el cuerpo docente. Para esta tarea **deberás adjuntar un README.md** en donde solamente se indiquen las **referencias utilizadas para su desarrollo**. Además, no existirá un des-descuento por un buen README.

7. Importante: Corrección de la tarea

La corrección de esta tarea es 100 % automatizada. 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 4: Tests](#)), en caso contrario se asumirá un resultado incorrecto.

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](#).

8. 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`.**
- Toda 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 prohibido. 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).