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

**IIC2233 Programación Avanzada (2018-2)**

# Tarea 01

## Entrega

- **Tarea**
  - **Fecha y hora:** 16 de septiembre de 2018, 23:59.
  - **Lugar:** GitHub — Carpeta: `Tareas/T01/`
- **README.md**
  - **Fecha y hora:** 17 de septiembre de 2018, 23:59.
  - **Lugar:** GitHub — Carpeta: `Tareas/T01/`

## Objetivos

- Determinar qué estructura de datos es mejor para alguna situación particular.
- Manejo y uso adecuado de estructuras de datos básicas incluidas en Python y sus operaciones.
- Manejar datos utilizando el paradigma de programación funcional.
- Aplicar la filosofía *"Do one thing and do it well"* creando varias funciones
- Reforzar el conocimiento de ignorar archivos en Git mediante el uso de `.gitignore`

# Índice

# 1. Introducción

Luego de que todo saliera mal con DCCorreos y el uso de los *error handlers*, el malvado Dr. $H^4$ , también conocido como Herny~~4444~~ Barba Negra, quiere recuperar sus *nebcoins* que se le olvidaron sacar antes de la caída de DCCorreos. Es por esto que te ha pedido un programa para poder obtener información de las aerolíneas, y así poder saber qué pasajes comprar con las pocas *nebcoins* que le van quedando, para revenderlos y ganar más criptomonedas.

Como el perverso Dr. $H^4$ se quedó sin tiempo por ~~tomar muchas ayudantías~~ tener muchos planes malvados, te pidió a ti también hacer las consultas que le permitirán seguir pagándole a ~~los ayudantes TPD de *Programación Avanzada*, a.k.a., sus *minions*~~ los constructores para así poder tener su nueva guarida. Es por esto que te ha encargado que puedas leer la **enorme**[1] base de datos que posee y poder hacer las consultas que se presentan más adelante.

Para lograr las metas que el malvado Dr. $H^4$ te ha pedido, deberás utilizar la base de datos que se entrega y hacer uso de programación funcional y estructuras de datos de Python de manera correcta y óptima. Además, por esta vez, el Dr. $H^4$ no quiere que utilices clases creadas con el *keyword* `class` debido a que son muy pesadas y cuesta mucho hacer inmutables los diccionarios creados en las **metaclases**[2] de los objetos.

# 2. *Cruncher Flights*

## 2.1. Lectura del archivo

Existen tres tipos de bases de datos: *small*, *medium* y *large*, cada una de mayor tamaño que la anterior. Cada base de datos consta de **tres** archivos con formato CSV separados por `","`, los que contienen información de aeropuertos, junto con los vuelos, pasajeros y sus respectivos viajes (`airports.csv`, `passengers.csv`, `flights.csv` y `flights-passengers.csv`).

Usando programación funcional, debes permitir el acceso a estas bases de datos, las cuales **no** pueden ser precargadas. La información debes generarla cuando es pedida, es decir, cuando se realiza una consulta.

A continuación se describirán las tablas a utilizar.

### 2.1.1. Pasajeros

La información acerca de los **pasajeros** se encuentre en el archivo `passengers.csv`, y cada fila representa un pasajero con su información. Cada columna del archivo representa lo indicado en la siguiente tabla:

| Nombre | Tipo de dato | Explicación |
|---|---|---|
| `id` | `integer` | Identificador único de cada pasajero. |
| `name` | `string` | Nombre completo del pasajero. |
| `class` | `string` | Clasificación económica del pasajero. |
| `age` | `integer` | Edad del pasajero. |

[1] Sí, realmente es enorme: tiene más de 80.000 líneas, por lo que una buena programación es bastante útil para disminuir los tiempos de consultas.

[2] Las metaclases, **quizás, eventualmente, algún día**, pueden ser útiles.

#### 2.1.2. Aeropuertos

La información acerca de los **aeropuertos** se encuentra en el archivo `airports.csv`, y cada fila representa un aeropuerto con su sigla. Cada columna del archivo representa lo indicado en la siguiente tabla:

| Nombre | Tipo de dato | Explicación |
|---|---|---|
| `icao` | `string` | Identificador único de cada aeropuerto. |
| `type` | `string` | Tipo de aeropuerto. |
| `lat` | `float` | Latitud del aeropuerto. |
| `long` | `float` | Longitud del aeropuerto. |
| `iso_country` | `string` | Código ISO del país del aeropuerto. |

#### 2.1.3. Vuelos

La información acerca de los **vuelos** se encuentra en el archivo `flights.csv`, y cada fila representa un vuelo con su información. Cada columna del archivo representa lo indicado en la siguiente tabla:

| Nombre | Tipo de dato | Explicación |
|---|---|---|
| `id` | `string` | Identificador único de cada vuelo. |
| `airport_from` | `string` | Identificador del aeropuerto de origen. |
| `airport_to` | `string` | Identificador del aeropuerto de llegada. |
| `date` | `datetime` | Fecha del vuelo. |

#### 2.1.4. Viajes

La información acerca de que pasajero voló en qué vuelo está contenida en el archivo `flights-passengers.csv`.

| Nombre | Tipo de dato | Explicación |
|---|---|---|
| `flight_id` | `string` | Identificador único de un vuelo. |
| `passenger_id` | `string` | Identificador del pasajero. |

Esto quiere decir que si en esta tabla tenemos una fila con los valores `(0,1)`, significa que el pasajero con `id` 1 viajó en el vuelo con `id` 0.

### 2.2. Consultas

#### 2.2.1. Formato de consulta

El formato de una consulta será un diccionario, cuya única *key* es el nombre de la consulta, y el valor será una lista de largo $k$ con los argumentos que requiere la consulta. Es decir:

```
{"nombre": [arg_1, arg_2, ... , arg_k]}
```

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

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

### 2.2.2. Consultas que retornan otra base de datos

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

1. `load_database(db_type: str) -> Generator`

   Esta consulta retorna un generador con la base de datos pedida.

   La base de datos retornada depende de la variable `db_type`, la cual puede tomar los valores (`"Passengers"`, `"Airports"`, `"Flights"` o `"Travels"`), en cada uno de esos casos debe retornar un generador con las personas, los aeropuertos, los vuelos o los viajes, respectivamente. Un ejemplo de esta consulta es:

   - `{"load_database": ["Airports"]}`

2. `filter_flights(flights: Generator, airports: Generator, attr: str, symbol: str, value: Union[datetime, int, float]) -> Generator`

   Esta consulta recibe un generador con vuelos, un generador con los aeropuertos, un atributo `attr` a comparar que puede ser `"date"` o `"distance"`, un símbolo de comparación (`"<"`, `">"`, `"=="` o `"!="`), y un valor para la comparación que puede ser una fecha o un número. Retornará un generador de vuelos que cumplan con la comparación. Algunos ejemplos de esta consulta son:

   - `{"filter_flights": [{"load_database": ["Flights"]}, {"load_database": ["Airports"]}, "distance", "<", 400]}`
   - `{"filter_flights": [{"load_database": ["Flights"]}, {"load_database": ["Airports"]}, "date", "!=", "2015-03-14 00:00:00"]}`

3. `filter_passengers(passengers: Generator, flights: Generator, travels: Generator, icao: str, start: datetime, end: datetime) -> Generator`

   Recibe un generadores con pasajeros, vuelos y viajes, el identificador único del aeropuerto de destino (`icao`) y un periodo de tiempo (`start`, `end`), donde debe ocurrir que `start <= end`. Retornará un nuevo generador con todos los pasajeros que viajen **a ese destino** entre las fechas entregadas. Un ejemplo de esta consulta es:

   - `{"filter_passengers": [{"load_database": ["Passengers"]}, {"load_database": ["Flights"]}, {"load_database": ["Travels"]}, "01WY", "2018-02-10 10:00:00", "2018-03-10 17:32:00"]}`

4. `filter_passengers_by_age(passengers: Generator, age: int, lower: bool) -> Generator`

   Recibe un generador con pasajeros y un `age` con valor entre 0 y 100. Si el booleano `lower` es `True`, retornará un nuevo generador con todos los pasajeros que tengan una edad menor al `age` entregado, sino entregará un generador con todos los pasajeros que tengan una edad mayor o igual. Si no se entrega un `lower`, este será por defecto `True`. Algunos ejemplos de esta consulta son:

   - `{"filter_passengers_by_age": [{"load_database": ["Passengers"]}, 25, False]}`
   - `{"filter_passengers_by_age": [{"load_database": ["Passengers"]}, 50]}`

5. `filter_airports_by_country(airports: Generator, iso: str) -> Generator`

   Recibe un generador con aeropuertos, un parámetro `iso` que representa el código del país en *string*.

   Para esta consulta, `iso` será un *string* que representa un código y la consulta debe retornar todos los aeropuertos que pertenezcan a dicho país.

   - `{"filter_airports_by_country": [{"load_database": ["Airports"]}, "US"]}`

6. `filter_airports_by_distance(airports: Generator, icao: str, distance: float, lower: bool) -> Generator`

   Recibe un generador con aeropuertos, un parámetro `icao` que representa el ICAO del aeropuerto como *string*, un parámetro `distance` que representa la distancia al aeropuerto en millas náuticas como `float` y finalmente el un parámetro `lower`, el cual es un `bool` que indica si queremos todos los aeropuertos cuya distancia es **menor** a la indicada de ser `True` o, en caso contrario, queremos los aeropuertos cuya distancia es **mayor** a la indicada.

   Por defecto, el valor de `lower` será `False`.

   - `{"filter_airports_by_distance": [{"load_database": ["Airports"]}, "SCJO", 3141.59, True]}`
   - `{"filter_airports_by_distance": [{"load_database": ["Airports"]}, "SCQM", 31415.92]}`

   Como se mencionó anteriormente, distancia entre aeropuertos se debe obtener en millas náuticas, ya que es el estándar internacional de aviación. Para obtener la distancia en millas náuticas se utilizara la fórmula de Haversine:[3]

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos(\varphi_1)\cos(\varphi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right)$$

   En donde:

   - $\varphi_1, \varphi_2$ son las latitudes de los puntos 1 y 2 respectivamente.
   - $\lambda_1, \lambda_2$ son las longitudes de los puntos 1 y 2 respectivamente.
   - $r$ es el radio de la tierra, aproximadamente 3440 millas náuticas.

   Para poder realizar estos cálculos, es altamente recomendable utilizar la librería **math** que es parte de la *standard library* de Python.

### 2.2.3. Consultas que no retornan otra base de datos

1. `favourite_airport(passengers: Generator, flights: Generator, travels: Generator) -> dict`

   Recibe un generador con pasajeros, otro con vuelos y otro con viajes, retorna un diccionario con *keys* igual a los identificadores de los pasajeros y *values* el código del aeropuerto al cual ese pasajero viajó más veces

   - `{"favourite_airport": [{"load_database": ["Passengers"]}, {"load_database": ["Flights"]}, {"load_database": ["Travels"]}]}`

[3] Para más información: `https://en.wikipedia.org/wiki/Haversine_formula`

2. `passenger_miles(passengers: Generator, flights: Generator, travels: Generator) -> dict`

   Recibe un generador con pasajeros, uno con vuelos y otro con viajes y retorna un diccionario que contiene a cada pasajero como *key* y las **millas acumuladas**[4] que ha recorrido como *value*. Algunos ejemplos de esta consulta son:

   - `{"passenger_miles": [{"load_database": ["Passengers"]},`
     `{"load_database": ["Flights"]},`
     `{"load_database": ["Travels"]}]}`
   - `{"passenger_miles": [{"filter_passengers_by_age": [{"load_database": ["Passengers"]},`
     `50, True]},`
     `{"load_database": ["Flights"]},`
     `{"load_database": ["Travels"]}]}`

3. `popular_airports(flights: Generator, airports: Generator, travels: Generator,`
   `topn: int, avg: bool) -> list`

   Esta consulta recibe generadores con vuelos, aeropuertos y viajes y retorna una tupla con los `topn` nombres de los aeropuertos más populares. Si el parámetro `avg` es `False`, el aeropuerto más popular es aquel que recibe a más gente. Sin embargo, si es `True`, el más popular es aquel que reciba más pasajeros **en promedio** por vuelo. Si no se introduce el parámetro `avg`, este será `False` por defecto. Un ejemplo de esta consulta es:

   - `{"popular_airports": [{"load_database": ["Flights"]},`
     `{"load_database": ["Airports"]},`
     `{"load_database": ["Travels"]}, 2]}`

4. `airport_passengers(passengers: Generator, flights: Generator, travels: Generator,`
   `icao1: str, icao2: str, operation: str) -> set`

   Esta consulta recibe generadores con pasajeros, vuelos y viajes, el identificador de dos aeropuertos **distintos** y un `operation` de consulta, el cual puede ser `"AND"`, `"OR"`, `"XOR"` o `"DIFF"`. Retornará un conjunto de los pasajeros que cumplan con las siguientes condiciones:

   a) Si `operation` es `"AND"`:
   Se retornan todas las personas que hayan viajado tanto por el aeropuerto `icao1` como por el aeropuerto `icao2`.

   b) Si `operation` es `"OR"`:
   Se retornan todas las personas que hayan viajado por el aeropuerto `icao1` o por el aeropuerto `icao2`.

   c) Si `operation` es `"XOR"`:
   Se retornan todas las personas que hayan viajado por alguno de los aeropuertos **excepto por** las personas que hayan viajado por ambos.

   d) Si `operation` es `"DIFF"`:
   Se retornan todas las personas que hayan viajado por el `icao1` que **no** han viajado por el `icao2`.

   Un ejemplo de esta consulta es:

   - `{"airport_passengers": [{"load_databse": ["Passengers"]},`
     `{"load_databse": ["Flights"]},`

[4] Calculada a partir de la fórmula de Haversine entre el aeropuerto de origen y el de llegada.

```
                                {"load_databse": ["Travels"]},
                                "01WY", "04IA", "AND"]}
```

Para efectos de consultas, se considerará que un pasajero ha estado tanto en su aeropuerto de despegue como aeropuerto de aterrizaje.

5. `furthest_distance(passengers: Generator, flights: Generator, travels: Generator, icao: str, n: int) -> list`

   Esta consulta recibe generadores con pasajeros, vuelos, y con los viajes, además, el código de un aeropuerto y un número `n`. Retorna una lista ordenada de forma decreciente con los $n$ pasajeros que pasaron por el aeropuerto cuyo destino final esta más lejos de este. Por defecto `n` es igual a 3. Algunos ejemplos de esta consulta son:

   - ```
     {"furthest_distance": [{"load_database": ["Passengers"]},
                            {"load_database": ["Flights"]},
                            {"load_database": ["Travels"]}, "04IA", 5]}
     ```
   - ```
     {"furthest_distance": [{"load_database": ["Passengers"]},
                            {"load_database": ["Flights"]},
                            {"load_database": ["Travels"]}, "01WY"]}
     ```

## 3. Interacción con consola

Para utilizar el programa, deberás implementar una interacción con la consola **mediante un menú** el cual **debe ser** a prueba de *inputs* inadecuados de todo tipo.

El menú debe estar implementado de tal manera que, al comienzo, se tenga la opción de elegir entre:

- Abrir un archivo con consultas, que se encuentre dentro de la carpeta del programa. Cada línea de este archivo tendrá una consulta válida.

  El usuario debe ser capaz de escribir el nombre de un archivo presente en la carpeta del programa y almacenar en memoria las consultas que aparecen en este. Luego podrá elegir una, varias o todas las consultas presentes en el archivo, y se deberán mostrar los resultados en la consola.

- Ingresar directamente una consulta por consola.

  La consola debe ser capaz de recibir las consultas de la manera en que se presentó en la sección 2.2.1.

  El usuario tendrá la opción de ingresar nuevas consultas, las cuales deben mostrar el resultado en pantalla. Luego, el usuario podrá guardar la consulta junto a su resultado en el archivo `output.txt`, siguiendo el formato establecido en 4.1.

- Leer el archivo `output.txt`.

  Por último, el usuario debe poder leer el archivo `output.txt`, donde se imprimirán en pantalla todas las preguntas y respuestas que posee y se tendrá la opción de eliminar algunas o todas las consultas del archivo.

Puede asumir que las consultas siempre estarán bien escritas, sin embargo, su programa no debe caerse si las opciones del menú son ingresadas erróneamente.

Por último, para cada sección dentro del programa, **se debe poder volver atrás** al menú anterior.

**Tip:** El menú puede funcionar con enumeración de las consultas para permitirle al usuario especificar cuál de ellas quiere realizar sin tener que escribirla por completo y así facilitar su uso.

### 3.1. Parseo de *input* y tratamiento de fechas

Debido a que las consultas deben ser ingresadas en *inputs* en formato de *string*, es altamente recomedable que para parsear las consultas desde el *input* utilices la librería `ast` y la función `literal_eval()` de esta librería, cuyo funcionamiento se encuentra **aquí**.

Puedes ocupar el siguiente *wrapper* de `literal_eval` —disponible en el módulo `iic2233_utils` incluído con el enunciado— que evita que el programa se caiga si se produce un error al parsear:

```python
def parse(string):
    """Parse a Python literal, without raising exceptions."""
    try:
        return literal_eval(string)
    except (ValueError, SyntaxError, TypeError):
        # ValueError is raised when an illegal node is in the tree
        # SyntaxError is raised when the syntax is not correct
        # TypeError is raised when a set contains a non-hashable object
        # Create an issue if you get another exception
        return None
```

Además, como podrás haber notado, los valores `datetime` de las consultas se encuentran en formato de *string*, por lo que se recomienda transformar estos *strings* a objetos del tipo `datetime` mediante la librería **`datetime`**, que viene con la librería estándar de Python. Además, considerando que la cantidad de datos con los que se trabajará es muy grande, es recomendable no guardar los objetos como objetos del tipo `datetime`, y encontrar una manera de representar la fecha como `int` para evitar uso de memoria innecesario[5].

## 4. Archivos relacionados a la tarea

**¡Esta sección es importante! No respetar esto afectará considerablemente la nota de su tarea por las repercusiones que tiene en las otras áreas.**

### 4.1. `output.txt`

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

[5] Los "ticks" del computador pueden servir.

```
---------- Consulta 1 ----------
CONSULTA 1 EN FORMATO 2.2.1
RESULTADO CONSULTA 1
---------- Consulta 2 ----------
CONSULTA 2 EN FORMATO 2.2.1
RESULTADO CONSULTA 2
...
---------- Consulta K ----------
CONSULTA K EN FORMATO 2.2.1
RESULTADO CONSULTA K
```

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

Sin embargo, es necesario que junto con el resultado de la consulta se imprima el `tipo` de output correspondiente. No será válido *hardcodear*[6]

# 5. Consideraciones

## 5.1. Programación funcional

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

1. Funciones generadoras
2. Expresiones generadoras
3. Contenedores (*e.g.* listas, diccionarios) por comprensión
4. `foreach`[7] —incluido en `iic2233_utils`—, con funciones que no alteren el *input*.

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

```python
# Este "for" no se puede utilizar.
processed = []
for flight in flights:
    processed.append(process(flight))

# Este "for" tampoco se puede utilizar.
for flight in flights:
    process(flight)

# Este "while" no se puede utilizar
result = True
while result:
    result = process(flight)

# Si la función process altera el input, entonces este "foreach" es ilegal
foreach(process, flights)

# Sin embargo, si la función no altera el input el "foreach" está permitido, como en este caso:
def save_flight(flight):
    with open('file.txt', 'a') as txt:
        txt.write(str(flight))
```

[6] Un ejemplo de *hardcodeo* sería imprimir que el output de `furthest_distance` es una lista directamente en vez de imprimir `type(output)`. [Ver **Wikipedia**]

[7] Como el que fue entregado en la actividad 02.

```python
foreach(save_flight, flights)

# En este caso, también está permitido el "foreach"
foreach(print, flights)

# Este "for" está permitido porque está en una función generadora
def generator_function(flights):
    for flight in flights:
        yield flight.data

# Este "for" está permitido porque está en una expresión generadora
first_elements = (x[0] for x in array_of_arrays)

# Este "for" está permitido porque está en un contenedor por comprensión
first_elements = {x[0] for x in array_of_arrays}
```

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

Dada la similitud en cómo se deben realizar ciertas búsquedas, para esta tarea se aplicará esta filosofía. Por lo tanto, es un requisito que su librería implemente funciones encargadas de hacer sólo una cosa. Esto significa que deben haber funciones pequeñas, que hagan algo específico y corto, que pueda ser generalizado y usado en otras partes del programa. En relación a esto, se evaluará que las funciones definidas no superen un **máximo de 15 líneas**, considerando la definición de la función (`def my_function(*args)`) como una de ellas.

Si consideras que alguna función tuya cumple con la filosofía, pero tiene un mayor largo de la cantidad permitida, **debes** justificar en el `README.md` por qué crees que sí cumple con *Do one thing and do it well*.

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

### 5.3. Uso de clases

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

### 5.4. `.gitignore`

Para no saturar los repositorios de GitHub, **deberás** agregar en tu carpeta `Tareas/T01/` un archivo llamado `.gitignore` de tal forma de **no** subir los CSV de la base de datos.

Este archivo se encarga de que los archivos no deseados (archivos muy grandes, bases de datos, entre otros) no se suban al momento de subir cambios a tu repositorio remoto.

**Tip**: Cuando uno hace `git add *` (agregar todo), el archivo `.gitignore` **no** se agrega, por lo que hay que agregarlo de manera explícita.

### 5.5. Limitaciones de materia

Queda **estrictamente prohibido** el uso de materia que no se ha visto en clases hasta la fecha de entrega del enunciado. Esto incluye conceptos como *threading*, *exceptions*, interfaces gráficas, entre otros.

## 6. 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 3.6.
- Si no se encuentra especificado en el enunciado, asume 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 archivo `README.md` **conciso y claro**, donde describas los alcances de tu programa, cómo correrlo, las librerías usadas, los supuestos hechos, y las referencias a código externo. **Tendrás hasta 24 horas después del plazo de entrega** de la tarea para subir el *readme* a tu repositorio.
- Tu tarea podría sufrir los descuentos descritos en la **guía de descuentos**.
- Entregas con atraso de más de 24 horas tendrán calificación mínima (1,0).
- Cualquier aspecto no especificado queda a tu criterio, siempre que no pase por sobre otro.

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