Tecnología Digital 1: Introducción a la Programación
Trabajo Práctico 2

Escuela de Negocios, UTDT - Primer semestre de 2024

El objetivo de este Trabajo Práctico es escribir programas para procesar y efectuar consultas sobre datos públicos de campanas verdes en la ciudad de Buenos Aires, como parte de un (imaginario) sistema informático usado para el gerenciamiento de los contenedores de la ciudad.

El archivo adjunto `campanas-verdes.csv` fue descargado el 08/05/2024 de la página web de Buenos Aires Data (`https://data.buenosaires.gob.ar/dataset/campanas-verdes`). Contiene información de casi 3000 campanas verdes (tachos de basura para material reciclable) en la ciudad, incluyendo la dirección, barrio, tipo de basura que se puede tirar y otros.

El objetivo de este Trabajo Práctico es escribir programas para modelar, procesar y efectuar algunas consultas sobre estos datos. En concreto, se pide:

1. Implementar una clase `CampanaVerde` que modele una campana verde individual en este problema y las consultas que se le pueden hacer. Debe tener al menos los siguientes atributos y métodos:

   - `CampanaVerde(...)`: construye un nuevo objeto de la clase `CampanaVerde`, con los parámetros que se consideren necesarios.
   - `c.direccion`: la dirección de la campana verde, de tipo `str`. Ejemplo: "`AGUIRRE 1447`".
   - `c.barrio`: el barrio donde está la campana verde, de tipo `str`.
   - `c.comuna`: la comuna donde está la campana, de tipo `int`. Ejemplo: 15.
   - `c.materiales`: un conjunto del tipo de materiales que se pueden depositar en la campana, de tipo `Set[str]`. Ejemplo: `{'Papel', 'Cartón'}`.
   - `c.latitud` y `c.longitud`: las coordenadas geográficas de la campana, ambas de tipo `float`.
   - `c.distancia(lat, lng)`: devuelve la distancia entre la `CampanaVerde` y el punto ingresado (punto dado por la latitud y la longitud con las cuales el método fue invocado). Para saber cómo calcular la distancia entre dos puntos geográficos, leer la sección **Cómo calcular distancias** al final de este enunciado.
   - `str(e)`: devuelve una representación como string de la campana verde `c` (esto se logra definiendo el método `__repr__`), que debe incluir entre los símbolos < y >: su dirección, una arroba, los materiales que se pueden depositar separados por el símbolo /, otro arroba 'y su barrio. Ejemplo: `'<AGUIRRE 1447@Papel/Cartón@CHACARITA>'`.

2. Implementar una clase `DataSetCampanasVerdes` que encapsule el concepto de una colección de datos de (muchas) campanas verdes y las consultas que se pueden hacer sobre ellas. Debe tener al menos los siguientes métodos:

   - `DataSetCampanasVerdes(archivo_csv)`: construye un dataset a partir del archivo CSV pasado como argumento. Por ejemplo, `DataSetCampanasVerdes('mis-datos.csv')` construye un objeto de la clase `DataSetCampanasVerdes` conteniendo los datos de las campanas verdes de un archivo llamado `mis-datos.csv`.
   - `d.tamano()`: devuelve la cantidad de campanas verdes en el dataset `d`. Debe tener complejidad temporal $O(1)$ en el peor caso.

- `d.barrios()`: devuelve el conjunto de todos los barrios existentes en el dataset `d`. Debe tener complejidad temporal $O(N * B)$ en el peor caso, donde N es la cantidad de campanas verdes y B la cantidad de barrios en `d`.
- `d.campanas_del_barrio(barrio)`: devuelve una lista con las campanas verdes (es decir, instancias de la clase `CampanaVerde`) del dataset `d` que están en el barrio indicado. Esta operación debe tener complejidad temporal $O(N)$ en el peor caso, donde N es la cantidad de campanas verdes en `d`.
- `d.cantidad_por_barrio(material)`: devuelve un diccionario que indica la cantidad de campanas verdes en cada barrio en las que se puede depositar el material indicado. Por ejemplo, para el dataset completo de `campanas-verdes.csv`, tomando `material`="Papel", retorna un diccionario que (entre otra información) contiene:[1]

  ```
  {...
    'AGRONOMIA': 25,
    'ALMAGRO': 91,
    'BALVANERA': 71,
  ...}
  ```

  La complejidad temporal de `cantidad_por_barrio` debe ser $O(N*B)$ en el peor caso, donde N es la cantidad de campanas verdes y B la cantidad de barrios en `d`.
- `d.tres_campanas_cercanas(lat, lng)`: devuelve una tupla con los tres objetos de tipo `CampanaVerde` que son las tres campanas verdes más cercanas al punto ingresado (punto dado por la latitud y la longitud con las cuales el método fue invocado). Para saber cómo calcular la distancia entre dos puntos geográficos, leer la sección **Cómo calcular distancias** al final de este enunciado. La complejidad temporal de `d.tres_campanas_cercanas(lat, lng)` debe ser $O(N)$ en el peor caso, donde N es la cantidad de campanas verdes.
- `d.exportar_por_materiales(archivo_csv, materiales)`: genera un nuevo archivo con nombre `archivo_csv` que contiene las campanas verdes en el dataset `d` en las que se pueda depositar todos los materiales del conjunto `materiales`, conjunto indicado como input del método. El archivo generado contiene únicamente las columnas `DIRECCION` y `BARRIO` .

Modo de entrega:

Se deben entregar los siguientes archivos, respetando los nombres con exactitud:

- **`campana_verde.py`**, con la definición de la clase `CampanaVerde`.
- **`campana_verde_test.py`**, con los tests de unidad de la clase `CampanaVerde`.
- **`dataset.py`**, con la definición de la clase `DataSetCampanasVerdes`.
- **`dataset_test.py`**, con los tests de unidad de la clase `DataSetCampanasVerdes`.
- **`informe.pdf`**, un documento en el cual se explique por qué cada método cumple el orden requerido. Incluir también cualquier aclaración adicional que se considere necesaria sobre cualquier parte del trabajo. Se espera que este documento sea conciso, de tres o cuatro páginas, y en formato PDF.
- Los archivos CSV necesarios para testear `DataSetCampanasVerdes`.

La carpeta adjunta `templates` contiene archivos de ejemplo para usar como referencia.

[1] Recordar que, al imprimir un diccionario, sus claves pueden aparecer en cualquier orden.

Observaciones:

- El trabajo se debe realizar en grupos de **tres personas**. La entrega consistirá en un trabajo original realizado íntegra y exclusivamente por las personas que integran el grupo. Se espera que las tres personas participen en toda la resolución. Se desaconseja fuertemente dividir y repartirse el trabajo.
- La fecha límite de entrega es el **domingo 23 de junio a las 23:55**. Los TPs entregados fuera de término serán aceptados hasta 48h más tarde, pero la demora incidirá negativamente en la nota.
- Los archivos deben subirse a la *Entrega del TP2* en la página de la materia en el campus virtual.
- El código entregado debe ejecutarse correctamente en `Python3`, y sólo pueden usarse elementos de Python vistos en clase. En particular, está permitido usar el método `ls.sort()` de listas. Ante la duda, consultar.
- Sólo está permitido importar las bibliotecas `unittest` (para los tests de unidad) y `csv` (para la lectura de archivos CSV; sugerimos usar la clase `csv.DictReader`).
- Es obligatorio especificar el tipo de todas las variables y funciones mediante sugerencias de tipos (*type hints*). Las funciones deben especificarse con el formato de *docstrings* visto en clase.
- Los archivos `campana_verde.py` y `dataset.py` no deben tener código principal; solamente definiciones de clases. El objetivo es que dichos archivos sean importados y usados por otros programas, como parte de un proyecto más grande.
- Durante el desarrollo y para el testing de unidad, se recomienda **evitar trabajar sobre el archivo CSV completo**. Por su tamaño, su lectura demora tiempo en cada ejecución y resulta difícil saber si los resultados son correctos. En cambio, conviene crear archivos CSV pequeños, conteniendo pocos registros tomados del archivo original, para tener mayor control de las ejecuciones.
- Puede suponerse que los objetos y sus métodos siempre se usarán de manera correcta. Es decir, si hay errores en la cantidad, tipo o valor de los argumentos, no se espera ningún comportamiento particular (la ejecución podría colgarse o terminar en un error, por ejemplo).
- **Sobre la evaluación del TP:** se tendrá en cuenta no solamente que el código sea correcto, sino también que sea claro, ordenado y modular, que esté comentado adecuadamente, que cumpla los órdenes de complejidad requeridos (con las justificaciones correspondientes), que el testing de unidad tenga un cubrimiento adecuado, que se respete con cuidado todo lo pedido en el enunciado, y que los archivos entregados sigan las indicaciones dadas.

Cómo calcular distancias:

Para calcular distancias, debe usarse la función `haversine`, que recibe dos puntos geográficos y devuelve la distancia entre ellos. Cada punto se representa como una tupla de dos floats (latitud y longitud). Opcionalmente, puede indicarse la unidad de distancia, que en nuestro caso será metros. Ejemplo de uso:

```
from haversine import haversine,Unit
punto1:tuple[float,float] = (-34.61315, -58.37723)
punto2:tuple[float,float] = (-36.69733, -56.68293)
haversine(punto1, punto2, unit=Unit.METERS)
```

El módulo `haversine` no está disponible por defecto en Python. Hay dos opciones para poder usarlo:

- (Recomendada) La opción más sencilla es copiar el archivo adjunto `haversine.py`[2] en la misma carpeta de los archivos `.py` del TP.
- (Avanzada) La otra opción es instalar el módulo `haversine` en Python. Cómo hacer esto depende de la plataforma (por ejemplo, en algunas hay que ejecutar "`pip3 install haversine`" en la consola del sistema operativo). Quienes se animen a instalarlo, pueden consultar en el foro. ¡No es difícil! Más info acá: `https://github.com/mapado/haversine`

[2] Fuente: `https://github.com/mapado/haversine`