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

IIC2233 Programación Avanzada (2020-1)

Tarea 00

Entrega

- **Tarea**
  - **Fecha y hora:** sábado 21 de marzo de 2020, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T00/`
- `README.md`
  - **Fecha y hora:** lunes 23 de marzo de 2020, 20:00
  - **Lugar:** Repositorio personal de GitHub — Carpeta: `Tareas/T00/`

Objetivos

- Aplicar competencias asimiladas en *Introducción a la Programación* para el desarrollo de una solución a un problema.
- Familiarizarse con el proceso de entrega de tareas y uso de buenas prácticas de programación.
- Procesar *input* del usuario de forma robusta, manejando potenciales errores de formato.
- Trabajar con archivos de texto para leer, escribir y procesar datos.
- Escribir código utilizando paquetes externos (*i.e.* código no escrito por el estudiante), como por ejemplo, módulos que pertenecen a la librería estándar de Python.

Índice

1. Introducción a DCCahuín

¡La era de las redes sociales llegó para quedarse, y por lo tanto, has decidido convertirte en *influencer*!

Sin embargo, es más difícil de lo que creías y sin importar lo que hagas, no logras que tus seguidores se multipliquen. Es por esto que decides utilizar tus conocimientos de Introducción a la Programación para crear tu propia red social y así convertirte en el usuario más famoso de esta. Así es como nace **DCCahuín**.

Figura 1: Logo de DCCahuín.

Todos queremos ser famosos y que nuestros *posts* sean vistos por todo el mundo. Lamentablemente esto genera mucho *spam* en los muros de los usuarios. Por esta razón, quieres que tu programa funcione en base a **seguidores**, por lo que tus *posts* publicados solamente serán vistos por aquellos usuarios que te sigan.

2. Manejo de *posts*

Los *prograposts* son la base de DCCahuín. Corresponden a textos breves escritos por los usuarios que serán mostrados a todos sus seguidores. Un usuario puede crear múltiples *prograposts*, pero un *prograpost* pertenece a un único usuario.

Cada *prograpost* debe tener una **fecha de emisión** (en el formato `yy/mm/dd`)[1], un **usuario creador** y un **cuerpo**, que corresponde al contenido de este. El texto puede incluir cualquier tipo de caracter, pero debe que tener al menos un caracter y no puede superar los 140.

Un usuario debe ser capaz de eliminar *prograposts* **creados por si mismo**. Si es que el usuario al momento de crear un *prograpost* no cumple los requerimientos mínimos mencionados en el párrafo anterior (cantidad de caracteres), se debe detener la operación e informarle al usuario del error en específico.

Finalmente, un usuario debe ser capaz de ver *prograposts*. Deberá tener la opción de ver solo aquellos publicados por si mismo, además de la opción de ver los *prograposts* publicados por los usuarios de los cuales es seguidor (esta segunda opción es conocida como tu **muro**). En ambas opciones anteriores debe ser posible ordenar los *posts* cronológicamente, tanto ascendente como descendentemente[2]. La información de los *prograposts* se almacena en el archivo `posts.csv`.

3. Seguidores

Cada usuario debe ser capaz de seguir a cuantos usuarios quiera, sin embargo nunca se podrá seguir a si mismo ni a un usuario que no exista. Además cada usuario podrá ser seguido por cualquier número de usuarios.

Seguir a otro usuario debe ser muy sencillo, basta con conocer su nombre en la aplicación y podrás seguirlo en el menú correspondiente. Al momento de seguir a otros usuarios podrás ver todos sus *posts* en tu muro, incluso los que fueron publicados antes de que seas su seguidor.

[1] Para obtener esto te recomendamos la librería `datetime`.
[2] Nunca está demás recordar las funciones de **ordenamiento** que posee Python.

Para saber qué usuarios siguen a quiénes al momento de ejecutar tu aplicación se debe consultar el archivo `seguidores.csv`.

4. Menús

Para que tu programa sea, ante todo, fácil e intuitivo de usar, decides hacer que la interacción por consola sea mediante **menús**. Cada menú muestra opciones disponibles al usuario, para luego recibir y procesar su decisión.

Todos los menús deben ser **a prueba de errores de usuario**, tener la opción de **volver atrás** o salir del programa cuando corresponda, y ser **fáciles** de utilizar. El formato de éstos queda a tu criterio, y puedes crear más submenús adicionales a los anteriores, siempre y cuando se cumpla con las funcionalidades mínimas.

Como mínimo deberás implementar los siguientes menús:

4.1. Menú de inicio

Este es el menú que verá el usuario al iniciar el programa. Debe preguntar primero si desea **iniciar sesión** con un nombre de usuario registrado, **registrar un usuario** o simplemente **salir**.

Para iniciar sesión debes pedir un nombre de usuario, el cual debes verificar que exista. Si el nombre ingresado es inexistente, deberá avisar al usuario y regresar al menú de inicio.

En el caso de registrar a un nuevo usuario, deberás pedir un **nombre**, el cual debes verificar que no esté registrado y que tenga extensión mínima de ocho caracteres alfanuméricos[3], de tal forma que posea por lo menos un número y una letra. En caso de que no se cumplan los requerimientos, deberá avisar al usuario y regresar al menú de inicio.

Una vez que se haya iniciado sesión o registrado un usuario correctamente, deberás permitir elegir acceder al **Menú de *prograposts***, al **Menú de seguidores** o **salir**.

Los nombres de usuario se encuentran en el archivo `usuarios.csv`, el cual debes actualizar cuando registres un nuevo usuario. Esto se encuentra explicado con mayor detalle en la sección Archivos entregados.

A continuación se muestra un ejemplo de como debería verse tu programa al iniciarse:

```
Bienvenido a DCCahuin!!
Seleccione una opción:
[1] Iniciar sesión
[2] Registrar usuario
[0] Salir

Indique su opción (0, 1 o 2): (input de usuario)
```

4.2. Menú de *prograposts*

Tu programa debe permitir las siguientes operaciones con *prograposts*:

1. Crear un *prograpost* siguiendo los requerimientos mínimos.
2. Eliminar un *prograpost*.
3. Ver *prograposts* creados por el usuario.

[3]Puedes buscar alguna función que sea de ayuda en la **biblioteca de métodos de `str`**.

4. Ver *prograposts* de los usuarios seguidos.
5. Volver al *Menú de inicio*.

Si se crea o elimina un *prograpost* exitosamente, este debe actualizar el archivo correspondiente, según lo explicado en la sección Archivos entregados. En el caso de que se elija la opción de ver *prograposts*, ya sea de los usuarios seguidos o los propios, el programa debe preguntar el orden en que estos se mostrarán (orden cronológico ascendente o descendente)[4]. Luego, se procede a mostrar todos los *prograposts* pedidos.

4.3. Menú de seguidores

Tu programa debe permitir las siguientes operaciones:

1. Seguir a un usuario.
2. Dejar de seguir a un usuario.
3. Volver al menú anterior.

Para seguir a otro usuario basta con ingresar el nombre de quien se quiere seguir, en caso de que el nombre ingresado no exista o se ingrese el nombre propio (no te puedes seguir a ti mismo) se le debe avisar al usuario y regresar al menú.

En el caso de dejar de seguir a otro usuario, basta con ingresar el nombre de quien se quiere dejar de seguir, en caso de que el nombre ingresado no exista o no se esté siguiendo, se le debe avisar al usuario y regresar al menú.

Cada vez que se cree o elimine una relación de seguimiento, se deberá actualizar el archivo `seguidores.csv`.

5. Archivos entregados

Los siguientes archivos contienen la base de datos de usuarios, seguidores y *prograposts* que usarás en tu programa. Cada vez que edites uno de estos archivos, **deberás mantener el formato especificado para dicho archivo.**

5.1. `usuarios.csv`

Este archivo contiene una lista de todos los usuarios registrados en DCCahuín. Al iniciar sesión debes verificar que el nombre ingresado se encuentre en este archivo. En el caso de que quieras registrar un nuevo usuario, debes escribirlo en este archivo si cumple con los requerimientos necesarios.

A continuación se muestra un ejemplo del archivo `usuarios.csv`:

```
JosephJoestar1
Desiaparición7
Elasticidad876
pYtHoN37
CapitánOMagma
Carrerin1313
```

[4] Para definir si el orden es ascendente o descendente puedes usar el parámetro `reverse` del método `sort` y la función `sorted`

5.2. `seguidores.csv`

Este archivo contiene la información de los seguidores de todos los usuarios. Es tu trabajo como programador que el archivo se actualice constantemente con la información de todos los usuarios de la aplicación.

En cada línea hay nombres de usuarios separados por comas (“,”) , el primer nombre de usuario representa al usuario **seguidor** y los demás son los nombres de los usuarios **seguidos** por el **seguidor**. Por lo tanto el archivo deberá poseer tantas filas como usuarios han sido creados

Un ejemplo del archivo `seguidores.csv` es el siguiente:

```
Carrerín,ElJefe,La Banda Elasticidad,Desiaparición,Capitán Magma
Capitán Magma,ElJefe
Desiaparición,ElJefe,Carrerín
La Banda Elasticidad,ElJefe,Carrerín,Capitán Magma
ElJefe
Joseph Joestar
```

En el ejemplo se puede ver que el usuario “El Jefe”no sigue a nadie, pero es seguido por cuatro usuarios. Además el usuario “Joseph Joestar”no es seguido por ningún usuario ni es seguidor de ningún usuario, sin embargo está de todos modos en el archivo.

5.3. `posts.csv`

Este último archivo contiene la información de todos los *prograposts* del sistema. Cada línea posee tres datos relevantes para un *prograpost* (**usuario creador**, **fecha de emisión** y **cuerpo**), separados por una coma (“,”).

Cabe notar que el cuerpo de un *prograpost* puede poseer comas, por lo que cualquier coma encontrada después de la que se utiliza para separar la **fecha de emisión** del **cuerpo** es parte de este último[5].

Un ejemplo del archivo `posts.csv` es el siguiente:

```
ElJefe,2019/12/23,Se viene la navidad y yo aqui programando #sad
Capitán Magma,2020/03/08,ayer fui a comprar leche, cereal, fruta para el desayuno... ñami!
Desiaparición,2020/02/24,¿alguien quiere ir al cine conmigo?
Joseph Joestar,2019/03/21,Tu siguiente linea será... ¡git push!
```

6. Buenas prácticas

Se espera que durante todas las tareas del curso, se empleen buenas prácticas de programación. Esta sección detalla dos aspectos que deben considerar a la hora de escribir sus programas, que buscan mejorar la forma en que lo hacen.

- **PEP8**:

  PEP8 es una guía de estilo que se utiliza para programar en Python. Es una serie de reglas de redacción al momento de escribir código en el lenguaje y su utilidad es que permite estandarizar la forma en que se escribe el programa para sea más legible[6]. En este curso te pediremos seguir un pequeño apartado de estas reglas, el cual puede ser encontrado en la **guía de estilos**.

[5] Podría ser útil el parámetro `maxsplit` del método `split` de *strings*.

[6] Símil a como la ortografía nos ayuda a estandarizar la forma es que las palabras se escriben.

- **Modularización**:

  Al escribir un programa complejo y largo, se recomienda organizar en múltiples módulos de poca extensión. Se obtendrá puntaje si ningún archivo de tu proyecto contiene más de 400 líneas de código.

Normalmente, estos dos aspectos son considerados como descuentos. A manera de excepción, para esta tarea serán parte del puntaje de la tarea, buscando que los apliques y premiando su correcto uso. Ten en cuenta que en las siguientes tareas, funcionarán como cualquier otro descuento de la sección Descuentos.

7. Descuentos

En todas las tareas de este ramo habrá una serie de descuentos que se realizarán para tareas que no cumplan ciertas restricciones. Estas restricciones están relacionadas con malas prácticas en programación, es decir, formas de programar que hacen que tu código sea poco legible y poco escalable (difícil de extender con más funcionalidades). Los descuentos tienen por objetivo que te vuelvas consciente de estas prácticas e intentes activamente seguirlas, lo cual a la larga te facilitará la realización de las tareas en éste y próximos ramos donde tengas que programar. Los descuentos que se aplicarán en esta tarea serán los siguientes:

- `README`: (1 décima)

  Se descontará una décima si no se indica(n) los archivos principales que son necesarios para ejecutar la tarea o su ubicación dentro de su carpeta. También se descontará una décima si es que no se hace entrega de un `README`. Esto se debe a que este archivo facilita considerablemente la recorrección de las tareas.

- **Formato de entrega**: (hasta 5 décimas)

  Se descontarán hasta cinco décimas si es que no se siguen reglas básicas de la entrega de una tarea, como son el uso de groserías en su redacción, archivos sin nombres aclarativos, no seguir restricciones del enunciado, entre otros[7]. Esto se debe a que en próximos ramos (o en un futuro trabajo) no se tiene tolerancia respecto a este tipo de errores.

- **Cambio de líneas**: (hasta 5 décimas)

  Se permite cambiar **hasta 20 líneas de código** por tarea, ya sea para corregir un error o mejorar una funcionalidad. Este descuento puede aplicarse si se requieren cambios en el código después de la entrega (incluyendo las entregas atrasadas). Dependiendo de la cantidad de líneas cambiadas se descontara entre una y cinco décimas.

- **Adicionales**: (hasta 5 décimas)

  Se descontarán hasta cinco décimas a criterio del ayudante corrector en caso de que la tarea resulte especialmente difícil de corregir, ya sea por una multitud de errores o por que el programa sea especialmente ilegible. Este descuento estará correctamente justificado.

- ***Built-in* prohibido**: (entre 1 a 5 décimas)

  En cada tarea se prohibirán múltiples funcionalidades que Python ofrece y se descontarán entre una a cinco décimas si se utilizan, dependiendo del caso. Para cada tarea se creará una *issue* donde se especificará qué funcionalidades estarán prohibidas. Es tu responsabilidad leerla.

[7] Uno de los puntos a revisar es el uso de *paths* relativos, para más información revisar el siguiente **material**.

- **Malas prácticas**: (hasta 5 décimas)

  Al igual que los *built-ins* prohibidos, también se prohibirán ciertas malas practicas y se descontarán entre una a cinco décimas si se realizan. Para cada tarea se creará una *issue* donde se especificará qué malas prácticas estarán prohibidas. Es tu responsabilidad leerla y preguntar en caso de tener dudas sobre las malas prácticas establecidas.

- **Entrega atrasada**: (entre 5 a 20 décimas)

  Las tareas serán recolectadas automáticamente y no se considerará ningún avance realizado después de la hora de entrega. Sin embargo, se puede optar por entregar la tarea de forma atrasada y se descontarán 5 a 20 décimas dependiendo de cuánto tiempo de diferencia haya entre la hora de entrega y la entrega atrasada.

- **Des-descuento**: (entre 1 a 5 décimas)

  Finalmente, se des-descontarán hasta 5 décimas por un `README` especialmente útil para la corrección de la tarea.

En la **guía de descuentos** se puede encontrar un desglose más específico y detallado de los descuentos.

8. `README`

Para todas las tareas de este semestre deberás redactar un archivo `README.md`, escrito en Markdown, que tiene por objetivo explicar su tarea y facilitar su corrección para el ayudante. Markdown es un lenguaje de marcado (como LaTeX o HTML) que permite crear un texto organizado y simple de leer. Pueden encontrar un pequeño tutorial del lenguaje en este ***link***.

Un buen `README.md` debe **facilitar la corrección de la tarea.** Una forma de lograr esto es explicar de forma breve y directa **qué cosas fueron implementadas, y que cosas no**, usualmente **siguiendo la pauta de evaluación**. Esto permite que el ayudante dedique más tiempo a revisar las partes de tu tarea que efectivamente lograste implementar, lo cual permite entregar un *feedback* más certero. Para facilitar la escritura del `README`, se te entregará una **plantilla** (*template*) a rellenar con la información adecuada.

Finalmente, como forma de motivarte a redactar buenos `README`s, todas las tareas tendrán **décimas de des-descuento** si el ayudante considera que tu `README` fue especialmente útil para la corrección. Estás décimas anulan décimas de descuento que les hayan sido asignadas hasta un máximo de cinco.

9. `.gitignore`

Cuando estés trabajando con repositorios, muchas veces habrán archivos y/o carpetas que no querrás subir a la nube. Por ejemplo, puedes estar trabajando con planillas de Excel muy pesadas, o tal vez estás utilizando un Mac y no quieres subir la carpeta `__MACOSX`, o el archivo `.DS_Store`, entre otros.

Una posible solución es simplemente tener cuidado con lo que subes a tu repositorio. Sin embargo esta "solución" es extremadamente vulnerable al error humano y podría terminar causando que subas muchos *gigabytes* de archivos y carpetas no deseados a tu repositorio.[8]

Para solucionar esto, `git` nos da la opción de crear un archivo `.gitignore`. Éste es un archivo **sin nombre, y con extensión `.gitignore`**, en el cual puedes detallar **archivos y carpetas a ser ignoradas por `git`**. Esto quiere decir que todo lo especificado en este archivo **no será subido a tu repositorio accidentalmente**, evitando los problemas anteriores.

[8] Lamentablemente basado en una historia real.

En esta ocasión el uso de este archivo **no será evaluado**, pero se recomienda su realización para que aprendan a crearlo y utilizarlo ya que será evaluado en las siguientes tareas.

Se recomienda utilizar el archivo `.gitignore` para ignorar archivos en tu entrega, específicamente el enunciado, los archivos indicados en Archivos entregados y todos los que no sean pertinentes para el funcionamiento de tu tarea. El archivo `.gitignore` debe encontrarse dentro de tu carpeta `TOO`. Puedes encontrar un ejemplo de `.gitignore` en el siguiente ***link***.

10. Importante: Corrección de la tarea

Para esta tarea, el carácter funcional del juego será el pilar de la corrección, es decir, **sólo se corrigen tareas que se puedan ejecutar**. Por lo tanto, se recomienda hacer periódicamente pruebas de ejecución de su tarea y `push` en sus repositorios.

Cuando se publique la distribución de puntajes, se señalará con color amarillo cada ítem que será evaluado a nivel de código, todo aquel que no esté pintado de amarillo significa que será evaluado si y sólo si se puede probar con la ejecución de su tarea. En tu `README` deberás señalar el archivo y la línea donde se encuentran definidas las funciones o clases relacionados a esos ítems.

11. 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 3.7.
- 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 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 48 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**. (con la excepción de PEP8 y Modularización, que tienen puntajes asociados)
- 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).