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

IIC2233 Programación Avanzada (2021-1)

Actividad Bonus

Entrega

- **Fecha y hora:** viernes 23 de julio de 2021, 20:00
- **Lugar:** Repositorio personal de GitHub — Carpeta: `Bonus/`

Importante: Nota que la carpeta de entrega no existe actualmente en tu repositorio personal. Por eso debes crear la carpeta `Bonus/`, al mismo nivel que las carpetas `Tareas/` y `Actividades/`, y hacer y entregar tu desarrollo en ella. De no entregar los archivos en el lugar especificado, no se corregirá la actividad.

Objetivos

- Tener la oportunidad de obtener bonificación en calificaciones del curso.
- Interactuar con una API obteniendo y enviando información para la elaboración de un programa.
- Utilizar y construir expresiones regulares para extraer información de textos, usando la biblioteca `pyrematch`.

1. Introducción

En esta actividad analizaremos el contenido de documentos Markdown a través de las herramientas que nos da la biblioteca `REmatch`. Para esto, utilizarás *webservices* para recuperar el contenido de un documento en formato *Markdown* desde una API desarrollada por el equipo docente. Necesitarás las bibliotecas `REmatch` y `requests`. Para instalarlas ejecuta:

```
pip3 install --upgrade pyrematch requests
```

En esta evaluación, tendrás que procesar documentos escritos utilizando el lenguaje de marcación Markdown. Este lenguaje se utiliza para organizar documentos en secciones, subsecciones y otras dimensiones menores, además de poseer sintaxis para demarcar listas, *checklists*, código, *links*, entre otros. Un ejemplo de documento *Markdown* es el siguiente, donde se destacan algunos de los elementos mencionados:

```
# Titulo

Un parrafo con un [link a la pagina del curso](https://iic2233.github.io/) y tambien
una *checklist* con:

* [X] Un elemento completado
* [ ] Un elemento sin completar

```

```
Una imagen:

<img src='https://picsum.photos/id/320/2689/1795' alt>

## Una subseccion

Con otros elementos, como `codigo` o bloques de codigo.

```python
variable = list()
otra_variable = 1
```

Tambien una lista no numerada con cosas:

* Cosa 1
* Cosa 2

Pero igual puede aparecer asi:

- Cosa 3
- Cosa 4
```

En este tipo de documento los títulos de las secciones se especifican escribiendo `# Título`, mientras que las subsecciones y subsubsecciones se demarcan con `##` y `###` respectivamente. Además, se pueden insertar imágenes utilizando un *tag* de HTML: `<img src='https://example.com/img.png' alt>`. Por otra parte, se puede insertar una sección de código utilizando el caracter `` ` ``, como se muestra en el ejemplo, donde en la primera línea se especifica el lenguaje del código y luego se indican las líneas de código. En un documento *Markdown* se pueden incluir listas numeradas (iniciando cada ítem con `1.`, `2.`, y así sucesivamente), no numeradas (iniciando cada ítem con `*` o `-`) y *checklists* (donde los elementos marcados se escriben `[X]` y los no marcados con `[ ]`), como se muestra en el ejemplo. Finalmente, se pueden agregar *links* directamente en el texto utilizando la sintaxis `[texto](link)`.

Para esta evaluación se ha dispuesto de una API, una aplicación que funciona en un servidor remoto, con la que deberás interactuar para completar la actividad. Puedes ver la documentación de la API desarrollada en tu navegador: https://actividad-bonus-iic2233.herokuapp.com/. Para acceder a los documentos *Markdown* deberás interactuar con la aplicación por medio del módulo `requests`, y además deberás entregar tu respuesta a cada consulta en la misma aplicación.

Archivos entregados

Se entregan tres archivos:

- `consultas.py`. Este archivo contiene la definición de los patrones y de la consultas según se especifica en la sección 3. **Debes completar este código, pero no debes cambiar los nombres de las constantes, ni los nombres o argumentos de las funciones a implementar**.
- `test.py`. Este archivo permite probar de manera automática el funcionamiento de las consultas en `consultas.py`.
- `api.py`. Este archivo contiene la definición de las funciones a completar según se especifica en la sección 4. **Debes completar este código, pero no debes cambiar los nombres de las**

constantes, ni los nombres o argumentos de las funciones a implementar.

- `documento.md`. Este archivo es un archivo *Markdown* de ejemplo, el mismo mostrado y utilizado a lo largo de este enunciado.

2. Usando `REmatch`

En esta evaluación usarás la biblioteca `REmatch` de expresiones regulares para extraer contenido como títulos de secciones y URLs desde un documento en *Markdown*. Tendrás que implementar **seis consultas**. Para cada una de ellas será necesario definir un patrón (expresión regular), compilarlo en `REmatch` para obtener un objeto `regex`, e iterar sobre los resultados obtenidos usando `regex.finditer(texto)`. Para cada resultado deberás acceder al contenido y la posición de un *match*.

Figura 1: Logo de REmatch

A modo de ejemplo, el formato general de la respuesta para cada consulta debería ser de esta manera:

```python
import pyrematch as re


def consulta_ejemplo(texto, patron):
    rgx = re.compile(patron)

    for match in rgx.finditer(texto):
        # Procesar los resultados
        pass

    return # Retornar los resultados procesados
```

3. Consultas

Las consultas deben escribirse en el archivo entregado `consultas.py`. Los patrones a utilizar se encuentran definidos en variables de nombre `PATRON_X`, donde `X` puede ser desde `1` hasta `6`. Dentro del mecanismo de corrección automática, cada consulta recibirá como argumento el `PATRON_X` correspondiente. **No modifiques los nombres de estas variables, ni las definas dentro de cada consulta, de lo contrario tu entrega no podrá ser evaluada**.

Cada consulta recibe dos argumentos: el patrón a aplicar (`patron`), y el texto sobre el cual se aplicará (`texto`). **El patrón se debe utilizar sobre el texto completo, no puedes realizar procesamiento adicional como separar por líneas, agrupar caracteres u otras funcionalidades de Python (revisa el método `consulta_ejemplo` explicado anteriormente)**. Como resultado, debe retornar una lista de `dict`, donde cada diccionario contiene dos llaves:

- **contenido**: el valor será el texto del *match* encontrado como `str`.

- **posicion**: el valor será un `array`[1] de dos elementos: la posición[2] de inicio y la posición de término del *match*.

A continuación se describen las consultas que debes implementar. Notar que cada función debe retornar sus resultados siguiendo el formato de resultado explicado anteriormente.

Consulta 1

`def consulta_1(texto, patron)`. En esta primera consulta, buscamos obtener el título del documento Markdown y la posición en donde aparece.

Un resultado parcial de esta consulta sobre el documento Markdown de ejemplo es:

```
[
    {
        "contenido": "Titulo",
        "posicion": (2, 8)
    }
]
```

Consulta 2

`def consulta_2(texto, patron)`. Ahora buscamos obtener todos los subtítulos y subsubtítulos del documento Markdown, junto a la posición en donde aparecen.

Un resultado parcial de esta consulta sobre el documento Markdown de ejemplo es:

```
[
    {
        "contenido": "Una subseccion",
        "posicion": (251, 265)
    }
]
```

Consulta 3

`def consulta_3(texto, patron)`. En markdown, podemos insertar imágenes mediante el uso de un *tag* HTML: `<img src='https://example.com/img.png' alt>`. Esta tercera consulta debe extraer la URL fuente de cada imagen presente en el documento, junto con su posición en el texto.

Un resultado parcial de esta consulta sobre el documento Markdown de ejemplo es:

```
[
    {
        "contenido": "https://picsum.photos/id/320/2689/1795",
        "posicion": (202, 240)
    }
]
```

[1] Como los valores se enviarán al servidor en formato JSON, da lo mismo si utilizas una tupla o una lista en Python, ya que al enviarse al servidor se serializarán y convertirán en un `array`. De todas formas, recomendamos tratar este valor como una `tuple`.

[2] Las posiciones serán los índices del primer y último carácter del *match* encontrados dentro del texto.

Consulta 4

`def consulta_4(texto, patron)`. En Markdown podemos insertar código de distintos lenguajes utilizando el caracter `` ` ``. Estos pueden presentarse dentro de un bloque de texto (utilizando solo un caracter por lado) o como un bloque o párrafo independiente con la siguiente sintaxis:

````
```python
variable = "Ejemplo"
```
````

En este caso, deberás entregar como contenido el lenguaje especificado en el bloque de código, pero el valor de posición es el **rango en el que se encuentra el contenido del bloque de código**, no el rango en que se encuentra el lenguaje.

Un resultado parcial de esta consulta sobre el documento Markdown de ejemplo es:

```
[
    {
        "contenido": "python",
        "posicion": (334, 370)
    }
]
```

Consulta 5

`def consulta_5(texto, patron)`. En esta consulta se deben extraer todos los ítems que ya estén marcados en una checklist, junto a sus posiciones.

Un resultado parcial de esta consulta sobre el documento Markdown de ejemplo es:

```
[
    {
        "contenido": "Un elemento completado",
        "posicion": (123, 145)
    }
]
```

Consulta 6

`def consulta_6(texto, patron)`. Finalmente, en esta consulta buscamos encontrar todos los links de la forma `` `[texto](referencia)` ``. Debes entregar todas las líneas de texto que posean un *link*, y la posición en el texto en la que se encuentra la referencia asociada a ese *link* (deberás utilizar dos variables).

Un resultado parcial de esta consulta sobre el documento Markdown de ejemplo es:

```
[{
        "contenido": (
            "Un parrafo con un [link a la pagina del curso]"
            "(https://iic2233.github.io/) y tambien"
        ),
        "posicion": (57, 83)
}]
```

Probando la solución

Para probar el funcionamiento de tus consultas, se entrega el archivo `test.py`, que contiene una clase `Test`. Esta clase ya se encuentra implementada, y puedes modificarla como desees para efectuar tus propias pruebas. Si bien esta clase no es parte de lo que debes completar, esperamos que te ayude durante tu desarrollo para probar tu código y para encontrar posibles errores. Para efectuar la corrección utilizaremos un archivo construido de la misma manera.

La clase `Test` recibe una lista de argumentos, una lista de *outputs* (resultados) esperados y el nombre de la consulta a probar. Una vez instanciada la clase, el método `probar_casos` recorre la lista de argumentos, utiliza la consulta a evaluar para obtener su *output*, y lo compara con el *output* esperado. Si ocurre una excepción durante alguno de estos pasos, el resultado se considera incompleto o incorrecto. Solo si el resultado obtenido coincide con el esperado, el test se considerará correcto.

4. Interacción con la API de la actividad

En esta evaluación utilizarás una API desarrollada por el equipo docente del curso. Utilizando los contenidos de *webservices*, deberás interactuar con la aplicación para registrarte, obtener documentos Markdown. La documentación completa se encuentra en el mismo sitio de la aplicación, https://actividad-bonus-iic2233.herokuapp.com/, pero a continuación dejamos un resumen de **algunas** de las rutas y recursos que deberás ocupar:

- **Agregar estudiante al curso** - `POST /estudiantes`: De acuerdo a la documentación de esta ruta, deberás utilizar el verbo `POST` para enviar un `JSON` con tu nombre y *username* en GitHub para inscribirte como parte del curso. Este paso es necesario para poder entregar respuestas durante el desarrollo de tu actividad.
- **Obtener un documento Markdown** - `GET /documentos/:identificador`: De acuerdo a la documentación de esta ruta, podrás obtener, por medio del verbo `GET`, cualquier documento Markdown utilizando un identificador numérico en la ruta. El valor de `:identificador` debe ser un número entero no negativo (por ejemplo: 1, 17, 420, etc.).
- **Probar respuesta a consulta sobre un documento** - `POST /estudiantes/:username/consultas`: De acuerdo a la documentación de esta ruta, deberás utilizar el verbo `POST` para enviar un `JSON` con el resultado que obtuviste al aplicar una consulta sobre un documento específico. El servidor comparará tu respuesta con la respuesta esperada, y te enviará un identificador de `proceso` que podrás utilizar para acceder a este *feedback* utilizando el verbo HTTP `GET` en la ruta **Recuperar los resultados del procesamiento**: `/estudiantes/:username/consultas/:proceso`.

Además de los métodos descritos, hay otras rutas que te servirán para desarrollar tu actividad. La descripción de todas las rutas se encuentra en el mismo sitio de la aplicación. Puedes utilizarlas para comprobar tu avance, pero la interacción con estas no será evaluada.

Para evaluar esta sección, se te entrega el archivo `api.py` con 3 funciones por completar. A continuación, se explica qué se debe realizar en cada función:

- `def registro(nombre: str, username: str) -> int`: Recibe el nombre y *username* de GitHub de quien se va a registrar en la aplicación. Deberás realizar la solicitud correspondiente al servidor, interactuando correctamente con la ruta para **Agregar estudiante al curso**. Deberás retornar un `int` con el *status code* (código de respuesta) obtenido al realizar la solicitud[3].

[3]Notar que cuando se crea un usuario por primera vez, este código debería ser `201 - Created`. Por otra parte, si ya se ha registrado un *username*, el código será `400 - Bad Request`

- `def descargar_documento(identificador_documento: int, ruta_documento: str)`: Recibe un identificador de documento (`identificador_documento`), el cual deberá utilizarse junto con la ruta para **Obtener un documento Markdown** para recibir un texto. Una vez que realices la solicitud, deberás guardar el texto en un archivo en la ruta `ruta_documento`. Este método no retorna nada, solo debe crear el archivo.
- `def entregar_consulta(n_consulta: int, identificador_documento: int, patron: str, respuesta: List[Dict]) -> str`: Recibe los argumentos necesarios para enviar los parámetros requeridos por **Probar respuesta a consulta sobre un documento**. Esto significa: el número de consulta, el identificador de documento, el patrón utilizado para obtener la respuesta y la respuesta misma (siguiendo el formato de salida descrito en la sección correspondiente a las consultas). Este método debe retornar el identificador del proceso retornado por el servidor.

5. Importante: Corrección de la tarea

La corrección de esta evaluación será automática. Esto quiere decir que el puntaje asignado depende directamente de si las funcionalidades están implementadas correctamente o no, y no pasará por la corrección detallada del equipo de ayudantes. Para lograr esto, se utilizarán archivos de *test* que ejecutarán tu código, y a partir de este, obtener un *output* definido. Si este *output* coincide con el esperado, se considerará ese *test* como correcto.

Para cada función, se realizarán múltiples y variados *tests*, de distinta complejidad, que buscarán detectar la correctitud de la implementación. Cada *test* se considerará correcto solo si no ocurre una excepción durante la ejecución de la función y si el resultado coincide con el esperado. Si todos los *test* de una función son correctos, se asignará puntaje completo asignado a la función. De la misma forma, se asignará la mitad del puntaje si al menos el 75 % de los *test* son correctos. En caso contrario, no se asignará puntaje a la implementación de esa función.

A continuación se listan las funciones que serán corregidas, y los puntajes asociados a cumplir el 100 % de sus *tests* automáticos:

- Funciones en `consultas.py`.
  - (1.0 pt.) `consulta_1`
  - (1.0 pt.) `consulta_2`
  - (1.0 pt.) `consulta_3`
  - (1.0 pt.) `consulta_4`
  - (1.0 pt.) `consulta_5`
  - (1.0 pt.) `consulta_6`
- Funciones en `api.py`.
  - (1.0 pt.) `registro`
  - (1.0 pt.) `descargar_documento`
  - (1.0 pt.) `entregar_consulta`

Una implementación completamente correcta otorga 9 puntos. Esto se traduce en una bonificación de décimas sobre el promedio de una de las áreas del curso: Actividades o Tareas. Se aplicarán sobre el promedio que sin bonificación sea **más bajo**. La cantidad de décimas será proporcional a los puntos obtenidos bajo la siguiente fórmula:

$$n = \begin{cases} \mathbf{P}/3 & \text{si } \mathbf{T} \leq \mathbf{AC} \\ \mathbf{P}/3 \times 2 & \text{si } \mathbf{T} > \mathbf{AC} \end{cases}$$

Donde n es la cantidad de décimas, **P** es la cantidad de puntos obtenidos en la corrección automática, **T** es el promedio ponderado acumulado de Tareas y **AC** es el promedio de Actividades, calculado como descrito en el programa y ***Syllabus***.

A modo de ejemplo, si los promedios de una persona en Tareas y Actividades son `3.90` y `4.75`, y obtiene `8.4` puntos en la corrección automática de esta evaluación, entonces sus promedios terminarían en `4.18` (`3.90 + 0.28`) y `4.75`, respectivamente. Notar que tal estudiante pasaría a aprobar el curso tras esta bonificación, ya que previamente no cumplía con el requisito de promedio mínimo en Tareas.

Consideraciones importantes de entrega

En esta evaluación **no** se recibirán entregas atrasadas. Cualquier *commit* ***pusheado*** posterior a la hora de entrega no se considerará. Se aconseja, como siempre, **realizar *commits* y *pushs* parciales de sus avances**. Por ejemplo, al terminar de implementar una función, es un buen momento para subir un avance, de manera que no acumulen muchos cambios nuevos al hacer *push* cerca de la hora de entrega.

Para esta evaluación **no se evaluará el uso de** `.gitignore`. De todas formas, lo único necesario que debes entregar son los **módulos a completar**: `api.py` y `consultas.py`. No pasa nada si se sube `test.py`.

Tampoco habrá entrega de `README`, ya que la corrección será automática. Recuerda seguir la estructura de archivos entregada y **no alterar los nombres las variables marcadas como `CONSTANTE` ni de las funciones a completar**, para asegurar estas puedan ser correctamente importadas.

Como mencionado al comienzo de este enunciado, se espera trabajes y entregues en tu repositorio personal, y en una carpeta **nueva** llamada `Bonus/`. Se espera que usen los módulos base siguiendo la misma estructura que en el código entregado, al primer nivel de la carpeta `Bonus/`:

```
Bonus/
├── api.py
├── consultas.py
└── test.py
```

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 3.7.
- Tu programa debe estar compuesto por uno o más archivos de extensión `.py`.
- En esta actividad solo es necesario utilizar los módulos `requests` y `pyrematch`, por lo que está prohibido utilizar módulos adicionales.

Las entregas que no cumplan con las restricciones del enunciado obtendrán la calificación mínima (0 puntos).