Pontificia Universidad Católica de Chile
Escuela de Ingeniería
Departamento de Ciencia de la Computación
IIC2233 — Programación Avanzada 2024-1

06 de Junio de 2024
Actividad

Actividad 5

Web Services y *Regex*

Entrega

- **Lugar:** Repositorio personal de GitHub — Carpeta:
  El código debe estar en la rama (*branch*) por defecto del repositorio: `main`.
- **Fecha máxima de entrega:** 10 de junio a las 20:00

Introducción: Yolanda Sultana

Gracias al Protocolo Kame-iSEKAI, se logró entablar comunicación con las tortugas del otro mundo, y luego de duros esfuerzos, se estableció el tratado *"Pepa is love, Pepa is life"*, donde se considera a cada tortuga como una persona con todos sus derechos y deberes. Han pasado varios meses y ahora se ven tortugas transitar por la calle, tortugas buscando trabajo, tortugas programando videojuegos, etc.

A raíz de todo lo anterior, una antigua amiga del DCC, Yolanda Sultana, te contacta porque contrató a una tortuga para que cree una API del Horóscopo (YolandAPI). Sin embargo, ella no se maneja en el consumo de APIs, así que te pide ayuda a ti a crear un programa que sea capaz de consumir la YolandAPI. Además, aprovechando tu buena voluntad, te pide integrar ciertas funcionalidades, ocupando expresiones regulares (*RegEx*) para validar y extraer información de textos; esto con el objetivo de ayudar a las tortugas que están aprendiendo sobre la cultura del planeta.

Archivos

En el directorio de la actividad encontrarás los siguientes archivos:

- Modificar `yolanda.py`: Contiene la clase `Yolanda` con todos los métodos a completar en esta actividad.
- No modificar `api.py`: Contiene todas las funcionalidades necesarias para levantar un servidor en el computador y poder ser consultado por `Yolanda` y los *tests*.
- No modificar `test.py`: Contiene el código necesario para ejecutar todos los *tests* relacionados a las consultas, modificaciones y uso de RegEx.

Estructura del programa

Esta actividad consta de tres partes, en las cuales se te pedirá que implementes métodos para aplicar los contenidos de *web services* y *regex*. Las 3 partes son independientes y presentan una cantidad de *tests*

distintas.

Parte 1 - Consulta a YolandaAPI

En esta parte, debes completar todos los métodos necesarios de la clase `Yolanda` para poder realizar diferentes consultas a YolandAPI. Para esto, deberás completar los siguientes métodos:

- Modificar `def saludar(self) -> dict:`
  Deberás realizar una *request* del tipo `GET` a la ruta `f"{self.base}/"` para obtener un mensaje de YolandAPI. Esta responderá con un JSON cuyo formato es el siguiente:

  ```
  {"result": "... mensaje personalizado ..."}
  ```

  Este método debe retornar un diccionario con 2 elementos: el *status code* de la consulta y el mensaje personalizado de YolandAPI contenido en `"result"`. El formato del diccionario a retornar es:

  ```
  {"status-code": ..., "saludo": ...}
  ```

- Modificar `def verificar_horoscopo(self, signo: str) -> bool:`
  Recibe un signo del horóscopo. Con este dato, deberás realizar una *request* del tipo `GET` a la ruta `f"{self.base}/signos"` para obtener todos los signos que posee YolandAPI en la base de datos. Esta responderá con un JSON cuyo formato es el siguiente:

  ```
  {"result": ["signo1", ..., "signoN"]}
  ```

  Luego, deberás verificar si entre dichos signos, se encuentra el `signo` consultado. Este método deber retornar un `booleano` que es `True` si es que el `signo` consultado existe entre los signos de YolandAPI y `False` en otro caso.

- Modificar `def dar_horoscopo(self, signo: str) -> dict:`
  Recibe un signo del horóscopo. Con este dato, deberás realizar una *request* del tipo `GET` a la ruta `f"{self.base}/horoscopo"` para obtener el horóscopo asociado a dicho signo. Para esto, deberás enviar un diccionario como *params* con el siguiente formado:

  ```
  {"signo": signo}
  ```

  YolandAPI responderá con un JSON cuyo formato es el siguiente:

  ```
  {"result": "mensaje personalizado al signo enviado...."}
  ```

  Este método deber retornar un diccionario con 2 elementos: el `status code` y el mensaje personalizado de YolandAPI contenido en `"result"`. El formato del diccionario a retornar es:

  ```
  {"status-code": ..., "mensaje": ...}
  ```

- Modificar `def dar_horoscopo_aleatorio(self) -> dict:`
  Deberás realizar una *request* del tipo `GET` a la ruta `f"{self.base}/aleatorio"`. Esta consulta puede llegar a entregar un enlace que deberás utilizar para realizar una segunda *request* del tipo `GET`. En ambas consultas, YolandAPI retornará un JSON con el siguiente formato:

  ```
  {"result": "..."}
  ```

  Este método debe retornar un diccionario con 2 elementos: el `status code` y la respuesta de YolandaAPI contenido en `"result"`. El formato del diccionario a retornar es:

  ```
  {"status-code": ..., "mensaje": ...}
  ```

  El contenido de `"status-code"` y `"mensaje"` dependerá de la respuesta obtenida en la primera consulta:

- Si el *status code* de la primera consulta **no** es 200, el `"status-code"` corresponderán al *status code* de la primera consulta y el `"mensaje"` será la respuesta enviada dentro de `"result"` por esta consulta.
- En otro caso, si el *status code* es 200, deberás utilizar el enlace obtenido en `"result"` y realizar una segunda *requests* del tipo `GET` a dicho enlace. Finalmente, el `"status-code"` corresponderá al *status code* de la segunda consulta y `"mensaje"` será la respuesta enviada por esta segunda consulta dentro de `"result"`.

Parte 2 - Modificar base de datos de YolandAPI

En esta segunda parte, seguirás trabajando con la clase `Yolanda`, pero ahora deberás hacer *requests* que permitan agregar, modificar y eliminar información de la base de datos.

Es importante saber que YolandAPI requiere *headers* de autenticación para cualquiera de las *requests* pedidas en esta parte. Para esto, cada método recibirá un **token de acceso de YolandAPI** que deberás incluir en la *request* para poder tener los permisos de modificar la base de datos.

Para YolandAPI, la forma de enviar la autenticación es incluir un diccionario como *header* que tenga el siguiente formato:

```
{
    'Authorization': 'valor de la llave',
}
```

A continuación, deberás completar los siguientes métodos de la clase `Yolanda` para interactuar con YolandAPI y su base de datos.

- Modificar `def agregar_horoscopo(self, signo: str, mensaje: str, access_token: str) -> str:`

  Recibe un signo del horóscopo, un mensaje y **token de acceso de YolandAPI**. Con estos datos, deberás realizar una *request* de tipo `POST` a la ruta `f"{self.base}/update"`, para agregar el signo a la base de datos de YolandAPI junto con el mensaje respectivo. Para esto, la *request* deberá utilizar el token para autenticarse y enviar un diccionario como `data` con el siguiente formato:

  `{"signo": signo, "mensaje": mensaje}`

  Tras realizar la *request*, YolandAPI retornará un JSON con el siguiente formato:

  `{"result": "..."}`

  Este método retornará un mensaje en función del `status_code` de la respuesta:

  - Si el `status_code` es 401, se retornará `"Agregar horóscopo no autorizado"`.
  - Si el `status_code` es 400, se retornará el mensaje contenido dentro del JSON en `"result"`.
  - En otro caso, se retornará `"La base de YolandAPI ha sido actualizada"`.

- Modificar `def actualizar_horoscopo(self, signo: str, mensaje: str, access_token: str) -> str:`

  Recibe un signo del horóscopo, un mensaje y **token de acceso de YolandAPI**. Con estos datos, deberás realizar una *request* de tipo `PATCH` a la ruta `f"{self.base}/update"`, para modificar el mensaje asociado a un signo en la base de datos de YolandAPI. Para esto, la *request* deberá utilizar el token para autenticarte y enviar un diccionario como `data` con el siguiente formato:

`{"signo": signo, "mensaje": mensaje}`

Tras realizar la *request*, YolandAPI retornará un JSON con el siguiente formato:

`{"result": "..."}`

Este método retornará un mensaje en función del `status_code` de la respuesta:

- Si el `status_code` es 401, se retornará `"Editar horóscopo no autorizado"`.
- Si el `status_code` es 400, se retornará el mensaje contenido dentro del JSON en `"result"`.
- En otro caso, se retornará `"La base de YolandAPI ha sido actualizada"`.

- Modificar `def eliminar_signo(self, signo: str, access_token: str) -> str:`
  Recibe un signo del horóscopo y **token de acceso de YolandAPI**. Con estos datos, deberás realizar una *request* de tipo `delete` a la ruta `f"{self.base}/remove"`, para eliminar un signo en la base de datos de YolandAPI. Para esto, la *request* deberá utilizar el token para autenticarte y enviar un diccionario como `data` con el siguiente formato:

  `{"signo": signo}`

  Tras realizar la *request*, YolandAPI retornará un JSON con el siguiente formato:

  `{"result": "..."}`

  Este método retornará un mensaje en función del `status_code` de la respuesta:

  - Si el `status_code` es 401, se retornará `"Eliminar signo no autorizado"`.
  - Si el `status_code` es 400, se retornará el mensaje contenido dentro del JSON en `"result"`.
  - En otro caso, se retornará `"La base de YolandAPI ha sido actualizada"`.

Parte 3 - *RegEx*

En esta última parte, debes completar 2 expresiones regulares que serán ocupadas para validar o extraer información de distintos textos.

Regex 1 - Validador de fechas

Debes completar la variable `regex_validador_fechas`, la cual corresponde a una expresión regular que valida que un texto completo cumple con el siguiente formato:

`"{día} de {mes} de {año}"`

donde:

- `día`: Corresponde a un número de hasta 2 dígitos. Puede ser cualquier combinación de estos.
- `mes`: Corresponde a una palabra de caracteres alfabéticos.
- `año`: Corresponde a un número de 2 o 4 dígitos. En caso de tener 2 dígitos, pueden ser cualquier número, y en caso de tener 4 dígitos, el año solo puede pertenecer al siglo XX o XXI.[1]
- Los distintos números y palabras pueden estar separador por cualquier tipo de espacio en blanco.

[1] El siglo XX corresponde a los años entre 1900 y 1999, mientras que el siglo XXI contiene los año entre 2000 y 2099.

A continuación, se entrega un listado de posibles casos que la expresión regular considera válidos e inválidos:

Casos válidos
`"1 de Enero de 2021"`
`"31 de DICIEMBRE de 22"`
`"15 de jul de 1998"`
`"02 de Feb de 23"`
`"10 de noviembre de 2020"`
`"99 de MesFalso de 2085"`
`"00 de otroMES de 99"`
`"0 de mayonesa de 2022"`

Casos inválidos
`"32 de enero de 2122"`
`"12 de junio de 1802"`
`"000 de diciembre de 2021"`
`"31 de octubre de 10000"`
`"29 de febrero de 021"`
`"5 de agosto, 2022"`
`"15, noviembre, 21"`
`"21 oct de 1956"`
`"12 marzo 22"`
`"7/julio/22"`
`"20-abril-3000"`
`"11 de 12 de 00"`
`"20 de de 2000"`
`"1 de de 2000"`

Regex 2 - Extractor de signos

Debes completar la variable `regex_extractor_signo`, la cual corresponde una expresión regular que extrae el signo del zodiaco de frases del tipo:

```
"Los {signo} pueden ___."
"Las {signo} pueden ___."
```

donde:

- `signo`: Corresponde a un sustantivo **en plural** que indica a las personas que pertenecen a un signo del zodiaco dado.
- `___`: Corresponde a cualquier texto que pueda ir después de la palabra "pueden".
- La frase termina con un punto.
- Las distintas palabras pueden estar separados por uno o más caracteres del tipo espacio en blanco.

Debes hacer un correcto uso de grupos para poder identificar el *substring* a extraer. Específicamente, se espera que el primer grupo corresponda al signo identificado.

A continuación, se entrega un listado de posibles casos válidos e inválidos, junto al signo extraído si corresponde:

Casos válidos	Signo extraído
`"Las Arianas pueden lograr un 7 en el ramo."`	`"Arianas"`
`"Los taurinos pueden esperar 4 puntos en la actividad."`	`"taurinos"`
`"Las GEMINIANAS pueden experimentar altibajos en los tests."`	`"GEMINIANAS"`
`"Los leoninos pueden encontrar un bug en el código."`	`"leoninos"`
`"Los SignoFalsianos pueden pasar el ramo con 8."`	`"SignoFalsianos"`

Casos inválidos	Causa
`"Los libra pueden recordar hacer git pull."`	Signo no está en plural.
`"les escorpianos pueden recordar hacer git add."`	La primera palabra no es "Las" o "Los".
`"Los sagitarianos recordarán hacer git commit."`	Falta la palabra "pueden" después del signo.
`"Las piscianas pueden pueden recordar hacer git push"`	Falta el punto final.

Notas

- **No puedes hacer *import* de otras librerías que no sean las ya entregadas en el archivo a completar.**
- Para esta actividad puedes seguir el orden que estimes conveniente.
- Recuerda que la ubicación de tu entrega es en **tu repositorio de Git**. En la rama (*branch*) por defecto del repositorio: `main`.
- Recuerda que esta evaluación presenta corrección **automatizada**. Si entregas un código que se cae al momento de correr los *tests*, será evaluado con 0 puntos.
- Además de los test, puedes probar tu código corriendo `api.py` en una ventana de terminal la cual se quedará a la espera, y luego `yolanda.py` en otra ventana de terminal con lo cual se imprimirán las últimas líneas del código.
- Si aparece un error inesperado, ¡léelo! Intenta interpretarlo y/o buscarlo en *Google*.

Ejecución de *tests*

En esta actividad se provee de `test.py` que contiene diferentes *tests* que ayudan a validar el desarrollo de la actividad. Para ejecutar este archivo, desde la terminal/consola debes escribir `python3 test.py` y se ejecutarán todos los *tests* de la actividad. Los puntos finales se asignarán según un conjunto de test privados.

En cambio, si deseas ejecutar un subconjunto de *tests*, puedes ejecutar lo siguiente en la terminal/consola:

- `python3 -m unittest -v test.VerificarConsultas`: para para ejecutar solo el subconjunto de *tests* que verifican las consultas a YolandaAPI de la Parte 1.
- `python3 -m unittest -v test.VerificarModificaciones`: para para ejecutar solo el subconjunto de *tests* que verifican las modificaciones a YolandaAPI de la Parte 2.
- `python3 -m unittest -v test.RegexTests`: para ejecutar solo el subconjunto de *tests* que verifican el correcto uso de expresiones regulares.

Importante: recuerda que si `python3` no funciona, probar con el comando específico de tu computador. Este puede ser `py`, `py3` o `python`.