



# Actividad B

## *Web Services y Regex*

### Entrega

- **Lugar:** En su repositorio privado de GitHub, en la **carpeta** Actividades/AB/
- **Hora del *push*:** 16:50

**Importante:** Antes de comenzar, comprueba que Git este funcionando correctamente en tu repositorio privado. Para esto, **sube los archivos base de la actividad de inmediato** (*add, commit, push*). Se espera que en esta actividad (así como en las demás actividades y tareas) utilices Git a lo largo de **todo tu desarrollo** como una herramienta, no sólo como un método de entrega. Es por esto que recomendamos enfáticamente que vayas subiendo tus cambios constantemente (*push*), ya que **problemas de último minuto** relacionados con la entrega y Git **no serán considerados**.

### 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 contrato a una tortuga para que cree una API del Horóscopo (YolandaAPI), pero 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 YolandaAPI. Además, aprovechando tu buena voluntad, te pide integrar ciertas funcionalidades, ocupando expresiones regulares (*regex*) para validar y extraer información de textos; 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_get.py`: Contiene el código necesario para ejecutar los *tests* relacionados a las consultas que no requieren autenticación. **Importante:** Por temas de estabilidad y compatibilidad

de los servidores para la API, los test tienen varios sleep, por lo que se demoran uno segundos en ejecutar.

- **No modificar** `test_autorizacion.py`: Contiene el código necesario para ejecutar los *tests* relacionados a las consultas que requieren autenticación. **Importante:** Por temas de estabilidad y compatibilidad de los servidores para la API, los test tienen varios sleep, por lo que se demoran uno segundos en ejecutar.
- **No modificar** `test.py`: Contiene el código necesario para ejecutar todos los *tests* de esta actividad. **Importante:** Por temas de estabilidad y compatibilidad de los servidores para la API, los test tienen varios sleep, por lo que se demoran uno segundos en ejecutar.

## 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 YolandaAPI. 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 YolandaAPI. 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 YolandaAPI 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 realiza una *request* del tipo GET a la ruta `f"{self.base}/signos"` para obtener todos los signos que posee YolandaAPI 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 YolandaAPI 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} 
```

YolandaAPI 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 YolandaAPI 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, YolandaAPI 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 YolandaAPI

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

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

```
1 {  
2     'Authorization': 'valor de la llave',  
3 }
```

A continuación, deberás completar los siguientes métodos de la clase `Yolanda` para interactuar con YolandaAPI 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 YolandaAPI**. 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 YolandaAPI junto con el mensaje respectivo. 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*, YolandaAPI 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 horoscopo 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 YolandaAPI 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 YolandaAPI**. Con estos datos, deberás realizar una *request* de tipo PUT a la ruta `f"{self.base}/update"`, para modificar el mensaje asociado a un signo en la base de datos de YolandaAPI. 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*, YolandaAPI 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 horoscopo 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 YolandaAPI ha sido actualizada".

- **Modificar** `def eliminar_signo(self, signo: str, access_token: str) -> str:`

Recibe un signo del horóscopo y **token de acceso de YolandaAPI**. 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 YolandaAPI. 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*, YolandaAPI 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 YolandaAPI 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.<sup>1</sup>
- Los distintos números y palabras pueden estar separados por cualquier tipo de espacio en blanco.

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

Casos válidos	Casos inválidos
"1 de Enero de 2021"	"32 de enero de 2122"
"31 de DICIEMBRE de 22"	"12 de junio de 1802"
"15 de jul de 1998"	"000 de diciembre de 2021"
"02 de Feb de 23"	"31 de octubre de 10000"
"10 de noviembre de 2020"	"29 de febrero de 021"
"99 de MesFalso de 2085"	"5 de agosto, 2022"
"00 de otroMES de 99"	"15, noviembre, 21"
"0 de mayonesa de 2022"	"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 a 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 separadas por uno o más caracteres del tipo espacio en blanco.

<sup>1</sup>El siglo XX corresponde a los años entre 1900 y 1999, mientras que el siglo XXI contiene los años entre 2000 y 2099.

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.