Pontificia Universidad Católica de Chile
Departamento de Ciencia de la Computación
IIC2233 - Programación Avanzada

Ayudantia Repaso Examen IIC2233

25 de Junio 2024

---

1. Repaso Midterm

1.1. Estructuras de Datos

1.1.1. Clasificación

- **Secuenciales:** Se guardan en memoria una al lado del otro (ordenadas), es decir podemos acceder a los datos por indice (`lista[0]`), junto con esto ademas podemos utilizar slicing para crear subestructuras (`lista[0:9]`), tenemos: tuplas, listas, colas, stacks
- **No secuenciales:** No se guardan lado a lado en memoria y por lo tanto no es posible utilizar slicing, tenemos: diccionarios, sets, namedtuples

1.1.2. Detalle de estructuras de datos

- **Tuplas:** Inmutables, Una vez creadas no es posible modificarlas, se utilizan preferentemente para persistencia de datos
- **Listas:** Mutables, pudes añadir y remover objetos
- **Stacks:** El ultimo elemento en entrar es el primero en salir
- **Colas:** El primer elemento en entrar es el primero en salir
- **Diccionarios:** Guarda la información en pares llave-valor, mucho mas eficiente que acceder a datos por indice
- **Sets:** funcionan como conjuntos matematicos y es muy eficiente para determinar si un elemento esta o no en el conjunto
- **Namedtuples:** Estructura de datos que "funciona como un objeto con atributos.[en] vez de acceder por indice se usa de la forma `namedtupla.valor`

1.1.3. *args y **kwargs

- `*args`**:** Para pasar una cantidad variable de argumentos posicionales.
- `**kwargs`**:** Para pasar una cantidad variable de argumentos por palabra clave.
- **Uso combinado:** Definir funciones que aceptan ambos tipos de argumentos para mayor flexibilidad, pero siempre deben ir los args y despues los kwargs
- **Desempaquetado:** Uso de `*` y `**` para desempaquetar listas/tuplas y diccionarios al llamar funciones.

1.2. Programación Orientada a Objetos avanzada

- **Herencia:** La herencia es una característica clave de la programación orientada a objetos, permite que una clase (subclase) herede atributos y métodos de otra clase, con esto facilita la reutilización de código.

- **Multiherencia:** La clase puede heredar de dos o más clases, al hacer esto se crea una jerarquía de clases (de izauierda a derecha) lo que al llamar un método del mismo nombre, Python usara esta jerarquía para decidir que método ejecutar.
- **super():** Con la función `super()` podemos hacer que python recorra toda la jerarquía de clases para ejecutar correctamente estos métodos.
- **Clases Abastractas:** Clases que sirven de modelo o guía para otras clases, su objetivo no es ser instanciadas sino ser heredadas. En ellas podemos definir métodos abstractos, los cuales DEBEN ser sobreescritos en todas las clases que heredan de ella
- **Properties:** permiten definir métodos que se comportan como atributos, proporcionando control sobre el acceso a los datos, tenemos:
  - **getter:** Retorna un valor, puede ser calculado en el método o un atributo privado
  - **setters:** Permite tener más control al reescribir un atributo, permitiendonos por ejemplo no salirce de ciertos margenes
  - **deleter:** Permite agregar comportamiento al eliminar un dato o atributo

1.3. Threading

- **Thread:** Objeto que permite ejecutar un trozo de codigo de manera paralela al programa `main`, para esto primero debemos darle una función objetivo e inicilizar el método con `thread.start()`
- **Timer (del threading):** Thread especifico que ejecuta un proceso solo una vez despues de un tiempo establecido
- **Locks:** Bloquea el acceso a información delicada, asegurandonos que solo un thread pueda acceder a este. Se debe asegurarse la liberación del Lock al ser utilizados o puede provocar un bloqueo general del programa (como en un taco automovilistico)
- **Events:** Los eventos nos permiten sincronizar distintos threads en su funcionamiento, pueden ser vistos como un semaforo, cuenta con los metodos:
  - `set()`: da la luz verde a los threads para avanzar
  - `clear()`: Pone la luz roja y detiene los threads
  - `wait()`: Bloquea el proceso del thread hasta que se de la luz verde
  - `is_set`: Retorna un booleano indicando si esta la luz verde o la luz roja

2. Interfaces Graficas

Las interfaces graficas es un componente escencial para que el usuario pueda ejecutar un programa de manera intuitiva.

2.1. PyQT

PyQT es la principal herramienta gráfica que ocuparemos, posee una cantidad extensa de widgets, algunos a destacar.

- Etiquetas (Qlabel, ...)
- Botones (QpushButton, ...
- Disposiciones (Layouts) ...

2.2. Patron de diseño front-back

- Para trabajar efectivamente con elementos graficos y de funcionalidad se crea una separación llamada FRONT y BACK end, en FRONT buscaremos desarrollar todo lo que tenga que ver con aspectos graficos e interactivos con el usuario, por su parte, en BACK buscaremos desarrollar funcionalidades que el FRONT pueda ejecutar

- Para que se comunique BACK y FRONT emplearemos SEÑALES y EVENTOS!.
  1. Se pulsa un boton en el FRONT, osea un **evento** ($QPushButton.clicked()$)
  2. Se envia la **señal** de que se presiono al Back
  3. El Back **procesa** la logica asociada a ese boton
  4. Envia una **respuesta** al FRONT
  5. El FRONT **despliega** una respuesta gracias al BACK.
- Todo lo anterior permite una alta cohesion (Cada una de las componentes del software debe realizar solo las tareas para las cuales fue creada) y un bajo acoplamiento (Cuando se modifique un componente se deben modificar todos los asociados a el), ademas de una modularidad, un alto escalamiento , una mantencion adecuada y diversas propiedades deseables en el entorno de trabajo Software.

2.3. Qthreads y Qtimer

Junto con la separación de roles, existira el concepto de buscar comportamientos ya sean repetivivos o aislados para asi poder generar diversos procesos al mismo tiempo (en base a nuestra funcion objetivo!), inclusive concurrencia. Esto lo lograremos con:

- QThreads: Permitira concurrencia tal que posee los mismos comportamientos que los Threads pero con metodos $.start()$ y $.stop()$.

  NOTA: De igual manera que con los Threads usuales, podremos ocupar LOCKS para las areas CRITICAS, esto se consigue con QMutex.
- QTimers: De igual manera que los QThreads permite concurrencia al generar comportamientos dado un cierto intervalo, misma logica que los QTimers solo que se ejecutaran de manera indefinida. Metodos asociado seran $.stop()$ para detenerlo, $.setinterval(N)$ para definir cada cuanto queremos que corra un hilo y $.start()$, $.stop()$.

  NOTA: Tambien podremos definir que se ejecute solo un Qtimer en caso que lo necesitemos con $SingleShot$, mas espeficiamente $.setsingleShot(True)$

2.4. Ejercicios!

22. ¿Cuál de las siguientes afirmaciones es **incorrecta** respecto a la separación *frontend-backend*?

    A) Esta separación busca una baja cohesión y alto acoplamiento.

    B) El *backend* y el *frontend* pueden estar en distintos lenguajes de programación.

    C) Según lo implementado en el curso, una responsabilidad de *backend* debería ser verificar que la contraseña ingresada por un usuario sea la correcta.

    D) Mostrar los *outputs* al usuario es responsabilidad del *frontend*.

    E) Un programa puede funcionar sin la necesidad de recurrir a esta separación.

Figura 1: Ejercicio Examen 2023-2

27. ¿Cuál de estos eventos está correctamente conectado?

    A)
    ```
    mi_qthread = QThread(...)
    mi_qthread.timeout.connect(funcion)
    ```

    B)
    ```
    mi_senal = pyqtSignal(...)
    mi_senal.clicked.connect(funcion)
    ```

    C)
    ```
    mi_qtimer = QTimer(...)
    mi_qtimer.timeout.connect(funcion)
    ```

    D)
    ```
    mi_boton = QPushButton(...)
    mi_boton.connect(funcion)
    ```

Figura 2: Ejercicio Compilado Midterm 2023-2

16. ¿Cuál afirmación es **correcta** sobre `Qthread` y `QTimer`?

    A) El `QThread` es la clase análoga a Thread, mientras que `QTimer` es el análogo a time. Por lo tanto, aplicamos concurrencia con `QThread` y usamos `.sleep` con el `QTimer`.

    B) Ambas son clases para realizar concurrencia, pero `QThread` se utiliza. exclusivamente cuando queremos aplicar herencia, mientras que `QTimer` se utiliza exclusivamente cuando queremos un método concurrente sin necesitar herencia.

    C) El `QThread` era la versión antigua para aplicar concurrencia, pero fue deprecado, es decir, ya no funciona, y solo se ocupa la nueva versión: `QTimer`.

    D) El `QThread` ejecuta de forma concurrente, mientras que el `QTimer` debe esperar a que no hayan otros `QTimers` corriendo para poder ejecutarse.

    E) El `QThread` ejecuta de forma concurrente la función indicada 1 vez, mientras que el `QTimer` permite llamar periódicamente a la función indicada.

Figura 3: Ejercicio Midterm 2023-2

18. Respecto a las diferencias que existen entre `QMainWindow` y `QWidget`, ¿cuál(es) de la(s) siguiente(s) afirmación(es) es/son **verdadera(s)**?

    I. `QMainWindow` permite incluir barra de menú, barra de herramientas y barra de estado, mientras que el `QWidget` no da todas estas opciones por defecto.

    II. `QMainWindow` permite que hayan múltiples ventanas abiertas en la aplicación, mientras que el `QWidget` no permite que hayan múltiples ventanas abiertas en la aplicación.

    III. `QMainWindow` puede contener un `QWidget` para desplegar contenido dentro de la ventana, mientras que el `QWidget` no puede contener otros *widget* dentro de la ventana.

    A) Solo I

    B) Solo III

    C) I y II

    D) I y III

    E) II y III

Figura 4: Ejercicio Midterm 2023-2

3. Programación Funcional

3.1. Iterables/Iterador

Vamos a tener 2 objetos principales, uno serán los iterables, osea aquellos objetos a los que puedo hacer un ciclo for y que por tanto tienen un iterador. Y los iteradores, que serán objetos distintos a los iterables, los cuales se encargan de poder recorrer el iterable, hay que tener en cuenta que se van consumiendo, osea que no puedo volver atras. Un iterable puede tener multiples iteradores recorriendolo de forma simultanea.

Método	**Iterable**	**Iterador**
__iter__	Devuelve un iterador	Se devuelve a sí mismo (return self)
__next__	No tiene	Devuelve el siguiente elemento del iterable. Si no quedan elementos, levanta una excepción StopIteration

Cuadro 1: Comparación entre Iterable e Iterador

3.2. Funciones sobre iterables

Funciones lambda: Funciones anónimas que no se necesitan ocupar en otros momentos, por lo que no se le asigna un nombre global.

map: Función que recibe uno o más iterables y una función. Aplica la función a los primeros elementos de los iterables, luego a los segundo, etc. Finalmente, devuelve un iterable con las respuestas obtenidas.

filter: Función que recibe un iterable y una función que devuelva True o False. Devuelve un iterable con solo los elementos que hayan devuelto True.

reduce: Función que requiere un iterable y una función. Busca reducir el iterable a un único elemento, por tanto aplica la función al primer elemento y al segundo, luego aplica la función a la respuesta anterior y al tercer elemento. Así sucesivamente.

3.3. Preguntas

24. ¿Cuál es el *output* del siguiente código?

```
data = [1, 2, 3]
print(reduce(lambda x, y: str(x) + str(y), map(lambda x: x * 2, data)))
```

A) 246

B) 12

C) 123

D) 642

E) 321

Figura 5: Ejercicio examen 2023-2

8. ¿Qué es lo **mínimo** que debes crear, para implementar una estructura **iterable** que se pueda recorrer directamente mediante el uso de `for`?

A) Una clase Iterable con el método `__iter__`, y una clase Iterador con el método `__next__`.

B) Una clase Iterable con el método `__iter__`, y una clase Iterador con los métodos `__iter__` y `__next__`.

C) Una clase Iterable con el método `__next__`, y una clase Iterador con el método `__iter__`.

D) Una clase Iterable con los métodos `__iter__` y `__next__`, y una clase Iterador con el método `__next__`.

E) Una clase Iterable con los métodos `__iter__` y `__next__`, y una clase Iterador con los métodos `__iter__` y `__next__`.

Figura 6: Ejercicio midterm 2023-2

4. Excepciones

Permiten manejar errores de manera sistematica, inclusive, permitiendo manejar ciertos errores de manera personalizada (¡Es una clase y por lo tanto heredable y personalizable!).

- **Raise** : Permite 'generar' excepciones al ocupar excepciones ya existentes (SyntaxError, ValueError, ...) en base a condiciones que generemos, INTERRUMPEN el flujo del programa.

```
def verificar_9(numeros):
    if numeros[0] != 9:
        raise ValueError("Formato equivocado")

numeros = "1524"
verificar_9(numeros)
print("No llegare aca!")
```

- **Try/Except**: Usadas usualmente en conjunto, try permite seguir el flujo usual del programa y en caso que ocurra algun tipo de excepción, se emplea except para capturar esa excepcion, de esta manera se permite PROSEGUIR el flujo del programa en caso de.

```
try:
    nombre = "Juan Perez"
    numero_asociado = int(nombre)

except ValueError:
    print("Error!")
```

4.1. Ejercicios!

2. Un programador utilizó un bloque de `try/except` para capturar una excepción del tipo `ValueError` y registrar dicho error en un archivo `.txt`. Ahora, además de registrar el error, desea volver a levantar la misma excepción con sus respectivos argumentos.

   ¿Cuál afirmación es **correcta** respecto a este caso?

   A) Una vez capturado el `ValueError`, no es posible levantar la misma excepción.

   B) Usando `print` se puede lograr este objetivo.

   C) Usando `raise` se puede lograr este objetivo.

   D) Si el bloque `try/except` está dentro de una función, con `return` se puede lograr el objetivo.

   E) Usando `emit` se puede lograr este objetivo.

Figura 7: Ejercicio Examen 2023-2

29. Considere el siguiente código:

```python
def favoritometro(series, nombre):
    if not isinstance(nombre, str):
        raise KeyError as "Nombre erróneo"
    else:
        print(f"Serie favorita: {series[nombre]}")

dict_series = {"Alice": "Breaking Bad", "Bob": "Dark"}
user = "uwu"
favoritometro(dict_series, user)
```

¿Qué error aparece al correr el código anterior?

A) `KeyError`

B) `TypeError`

C) `SyntaxError`

D) `ValueError`

Figura 8: Ejercicio Compilado MidTerm 2023-2

5. Bytes y serialización

5.1. Manejo de bytes

Los bytes son secuencias de 8 bits, y cada bit es un 0 o un 1, por lo que un byte puede almacenar 2^8 combinaciones de 0's y 1's. Cuando armamos secuencias de bytes podemos representar distintos caracteres, según el encoding que estemos usando (ascii, utf-8, etc). La representación de bytes en python es principalmente a través de dos objetos: el objeto bytes que se comporta como un string ya que es inmutable, y el bytearray que se comporta como una lista ya que podemos modificarlo.

5.2. JSON y pickle

Estas dos librerías nos permiten transformar un objeto en una serie o secuencia de bytes, que pueden ser comunicadas hacia otro programa y son especialmente útiles en el contexto de networking. Las principales características de cada uno de los dos métodos son:

- JSON: Sólo permite serializar int, str, float, dict, bool, list, tuple y NoneType. Puede ser leído por programas en lenguajes distintos a Python. Al serializar entrega un string que es legible, y luego puede ser encodeado a bytes.
- Pickle: Permite serializar cualquier objeto de python, incluso objetos creados por nosotros. Al serializar entrega bytes, que no son legibles. Tiene riesgos de inyección de código.

5.3. Preguntas

2. Respecto a JSON y *pickle*, es **correcto** afirmar que:

   I. *pickle* puede serializar más tipos de datos de los que puede JSON.

   II. Al serializar con JSON o *pickle* se obtienen *bytes* como resultado.

   III. *pickle* es la mejor opción para comunicarse con otros *webservices*.

   A) Solo I.

   B) Solo II.

   C) Solo III.

   D) I y III.

   E) I, II y III.

Figura 9: Ejercicio Compilado Examen 2023-2

3. Respecto a los *bytearrays*, es **correcto** afirmar que:

   A) Pueden ser iterados, pero no se puede acceder a un índice específico.

   B) Se le pueden agregar números entre 0 y 256.

   C) Al igual que los *bytes*, son estructuras inmutables.

   D) A diferencia de los *bytes*, son estructuras mutables.

   E) Una vez instanciado el *bytearray*, se pueden modificar los datos dentro, pero no agregar o quitar elementos.

Figura 10: Ejercicio Compilado Examen 2023-2

6. Networking

6.1. Elementos básicos

El networking es un concepto dentro de la computación que nos permite la comunicación entre dos o más computadores. Para esto, necesitamos conocer los siguientes conceptos.

- **Dirección IP:** Un número con el que se identifica a un *host* en particular. Puede ser en formato IPv4 (2^{32} valores posibles) o IPv6 (2^{128} valores posibles).
- **Puertos:** Vía de comunicación dentro de un computador. Un computador tiene varios puertos y cada uno se ocupa para solo una aplicación.
- **Protocolos de comunicación:** Protocolos que permiten que el mensaje sea emitido y recibido. Los más importantes son TCP y UDP.
- **Protocolo UDP:** Garantiza la rapidez por sobre la llegada correcta de toda la información. Por ejemplo, un video pixelado en YouTube.
- **Protocolo TCP:** Garantiza la llegada correcta de todos los datos por sobre la rapidez, por lo que se necesita un *handshake* donde emisor y receptor establecen los parametros de la conexión. Por ejemplo, subir una tarea a GitHub.

6.2. Preguntas

1. En el contexto de un cliente y servidor que se comunican a través de *networking*, ¿cuáles de las siguientes acciones son las **mínimas** que se deben implementar para asegurar un **correcto** manejo de los mensajes?

   I. Codificación

   II. Decodificación

   III. Encriptación

   IV. Desencriptación

   A) I y II

   B) I y III

   C) II y IV

   D) III y IV

   E) I, II, III y IV

Figura 11: Ejercicio Examen 2023-2

5. Dado el caso de un cliente que se desea conectar a un servidor mediante *networking*, ¿cuál de las siguientes afirmaciones es **correcta** respecto al siguiente código?

```
1 sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
2 sock.connect(("localhost", 3000))
```

   A) El argumento `"localhost"` indica la dirección IP en que se está ejecutando el **servidor**, y el argumento `3000` indica el puerto que está usando el **servidor** para la comunicación.

   B) El argumento `"localhost"` indica la dirección IP en que se está ejecutando el **cliente**, y el argumento `3000` indica el puerto que está usando el **cliente** para la comunicación.

   C) El argumento `"localhost"` indica la dirección IP en que se está ejecutando el **servidor**, y el argumento `3000` indica el puerto que está usando el **cliente** para la comunicación.

   D) El argumento `"localhost"` indica la dirección IP en que se está ejecutando el **cliente**, y el argumento `3000` indica el puerto que está usando el **servidor** para la comunicación.

   E) El argumento `"localhost"` indica la dirección IP en que se está ejecutando tanto el **servidor** como el **cliente**, y el argumento `3000` indica el puerto que está usando tanto el **servidor** como el **cliente** para la comunicación.

Figura 12: Ejercicio Examen 2023-2

7. Webservices

El protocolo HTTP sirve para pedir y enviar datos a una página web. El protocolo consiste en una solicitud y una respuesta. La solicitud es realizada por el cliente, a lo cual el servidor responde con información.

7.1. Cliente

Al hacer una sólicitud, se envían los siguientes datos:

La URL: La URL es la dirección donde se ejecuta la consulta. Está compuesta de un esquema (http o https), un dominio (www.youtube.com), una ruta (/watch) y los query params (?v=dQw4w9WgXcQ). Juntando las partes del ejemplo, tenemos `https://www.youtube.com/watch?v=dQw4w9WgXcQ`.

El método: El método se utiliza para especificar qué tipo de solicitud de quiere hacer. Los significados son una convención, pero se recomienda fuertemente apegarse a sus significados. Los métodos más comunes son:

- GET: pide uno o varios elementos existentes
- POST: crea un nuevo elemento con la información enviada
- PATCH: realiza un update parcial
- PUT: reemplaza un elemento en su totalidad
- DELETE: borra un elemento

Los headers: Agregan meta-información sobre la solicitud. Puede ser, en qué formato está codificado el cuerpo, quién es el usuario realizando la solicitud, etc.

El cuerpo: Es la información central de la solicitud. Por lo general contiene la información que será creada / editada en una solicitud POST / PATCH / PUT.

Para hacer consultas desde python, utilizamos el módulo requests, que nos provee de todo lo necesario para hacer consultas. Por ejemplo, para hacer una solicitud patch que actualize el nombre de la persona, y que contiene el cuerpo en formato JSON, un posible código que haga eso es el siguiente:

```
import requests

headers = {
    "content-type": "application/json"
}

body = {
    "nombre": "Pepito"
}

requests.patch(f"{BASE_URL}/personas/{id}", headers=headers, json=body)
```

Veremos en la siguiente parte lo que retorna el servidor

7.2. Servidor

El servidor debe responder al menos con algo de información en el body y con un código de estado. También puede entregar headers para entregar más información sobre la solicitud.

Códigos de estado: Indican qué fue lo que pasó al ejecutar la solicitud. Los más importantes son:

- 2XX: Todo salió bien, no hubo error
- 4XX: El cliente se equivocó, es decir, no mandó credenciales válidas, mandó un cuerpo que no es aceptado, envió un método HTTP no aceptado por el servidor, etc.
- 5XX: El servidor tuvo un error, es decir, algo falló en el servidor y por lo tanto la respuesta no debe tomarse en cuenta.

Cuerpo: Puede estar en formato JSON, en texto plano, u otros. Depende del servidor qué es lo que se entrega acá.

Headers: Siguen la misma lógica de los headers del cliente.

7.3. Ejercicios:

14. En el contexto de las consultas que se realizan por medio una API, ¿cuál o cuáles elementos de una *response* no están presentes en una *request*?

    I. *Status code*

    II. *Header*

    III. *Params*

    IV. *Body*

    A) Solo I

    B) Solo III

    C) Solo IV

    D) I y IV

    E) I, II, III y IV

15. ¿Cuál de la siguientes alternativa relaciona **incorrectamente** el caso descrito con el grupo de códigos de estado HTTP que respondería una API?

    A) Solicitar el horóscopo para `"Acuario"` y que la API responda con el texto asociado a dicho signo. Grupo de códigos de estado HTTP: `2XX`.

    B) Solicitar el horóscopo para `"Siuu"` y que la API responda que no encuentra dicho signo en la base de datos. Grupo de códigos de estado HTTP: `4XX`.

    C) Agregar un nuevo signo a la base de datos, pero la API responde que no tengo permisos para realizar dicha acción. Grupo de códigos de estado HTTP: `5XX`.

    D) Solicitar el horóscopo para `"Siuu"` y que la API se cae por un error interno. Grupo de códigos de estado HTTP: `5XX`.

    E) Cambiar el horóscopo para `"Sagitario"` por otro texto y que la API responde que el texto fue actualizado correctamente. Grupo de códigos de estado HTTP: `2XX`.

16. Si se decide utilizar un *endpoint* de una API que cumple el protocolo HTTP, ¿cuál de los siguientes métodos no modifica la base de datos de la API?

    A) `PUT`

    B) `POST`

    C) `PATCH`

    D) `GET`

    E) `DELETE`

17. Se dispone un servidor con la siguiente API:

```python
class API:
    def __init__(self):
        self.database = [{"id": "A", "nombre": "Anya"},
                         {"id": "B", "nombre": "Yor"}]

    def metodo_a(self, id):
        personas = list(filter(lambda d: d["id"] == id, self.database))
        persona = personas[0]
        return persona

    def metodo_b(self, id, nombre):
        self.database.append({"id": id, "nombre": nombre})

    def metodo_c(self, id, nombre):
        personas = list(filter(lambda d: d["id"] == id, self.database))
        persona = personas[0]
        persona["nombre"] = nombre
```

Cuando se hace una solicitud, al servidor, del tipo `POST`, `GET` o `PATCH`, se ejecutará el método correspondiente al tipo de solicitud. Puedes asumir que cada método está asociado a solo un tipo de solicitud distinta y que cumplen con el protocolo HTTP.

Considerando la base de datos actual (`self.database`) que contiene 2 elementos, ¿cuál afirmación es **incorrecta** sobre este servidor y su API?

A) Hacer una solicitud del tipo `GET` con el siguiente dato: `id="A"`, hará que la API retorne `{"id": "A", "nombre": "Anya"}`.

B) Hacer una solicitud del tipo `GET` con el siguiente dato: `id="F"`, hará que la API retorne un código de estado HTTP del grupo `5XX`.

C) Hacer una solicitud del tipo `POST` con los siguientes datos: `id="H"` y `nombre="Nightfall"` hará que se incluya un nuevo dato a la base de datos.

D) Hacer una solicitud del tipo `PATCH` con los siguientes datos: `id="A"` y `nombre="Sylvia"` hará que el dato con `id="A"` modifique su nombre por `"Sylvia"`.

E) Hacer una solicitud del tipo `POST` con los siguientes datos: `id="A"` y `nombre="Loid"` hará que el dato con `id="A"` modifique su nombre por `"Loid"`.

8. Regex

Las expresiones regulares, o RegEx, son expresiones que permiten la búsqueda de patrones dentro de strings, de forma eficiente. Los siguientes caracteres especiales componen las expresiones regulares:

- [ ]: Permiten especificar clases de caracteres
- $+$: Permite indicar que una expresión se puede repetir una o más veces
- $*$: Permite indicar que una expresión se puede repetir una o más veces
- ?: Permite que la expresión anterior pueda estar o no
- $\{m, n\}$: Permite indicar que la expresión anterior se repita entre m y n veces
- . : Permite especificar un match con cualquier caracter excepto salto de linea
- ^: Permite especificar el inicio de un string
- $: Permite especificar el final de un string
- (): Permite especificar un grupo
- $A|B$: Espeficia un 'or', es decir, que alguno de A o B se debe cumplir
- \: Permite escapar un meta-caracter para especificar su símbolo en vez de su significado

Además, los métodos disponibles en la librería *re* de python son los siguientes:

- re.match: Verifica si un substring cumple con la expresion regular desde el inicio del string

- re.fullmatch: verifica si el string completo cumple con la expresión regular
- re.search: Verifica si algun substring cumple con la expresión regular
- re.sub: Permite reemplazar un patrón por otra secuencia de caracteres en un string
- re.split: permite separar un string de acuerdo a un patrón

8.1. Ejercicios:

9. ¿Cuál o cuáles de las siguientes acciones pueden realizarse mediante el módulo `re` para expresiones regulares de Python?

   I. Verificar si algún *substring* de un texto cumple con la expresión regular.

   II. Listar todos los *substrings* de un texto que cumplan con una expresión regular.

   III. Reemplazar todos los *substrings* de un texto que cumplan con una expresión regular por un texto diferente.

   A) Solo I

   B) Solo II

   C) I y II

   D) I y III

   E) I, II y III

10. Respecto a las expresiones regulares, es **incorrecto** afirmar que:

    A) Son secuencias especiales de caracteres que permiten buscar *strings* dentro de otro *string*.

    B) Poseen una sintaxis específica para poder ser expresiones regulares válidas.

    C) Poseen un conjunto de caracteres especiales para definir patrones más generales. Por ejemplo: `.`, `$`, `?`.

    D) Una limitación que tienen, es que no pueden encontrar los caracteres especiales dentro de otro *string*. Por ejemplo, no puede encontrar `.`, `$` o `?`.

    E) Una limitación que tienen, es que no pueden capturar patrones que tengan una cantidad aleatoria de caracteres.

12. En expresiones regulares usamos: `()` para agrupar, `*` para indicar que el carácter o grupo está 0 o más veces y `|` para definir un *or*. Dado lo anterior, ¿**cuántos** *substrings* hacen *match* entre la Frase y la Expresión?

    Frase: `'¡Aaahhh! La cama de mi casa está ocupada por mi gata'`
    Expresión: `r'a(s|m)*a'`

    A) 3

    B) 5

    C) 0

    D) 4

    E) 2