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

Ayudantia Repaso Examen IIC2233

26 de Noviembre 2024

---

1. Repaso Midterm

1.1. Estructuras de Datos

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

Detalle de estructuras de datos

- **Tuplas:** Inmutables, Una vez creadas no es posible modificarlas, se utilizan preferentemente para persistencia de datos
- **Listas:** Mutables, puedes añadir y remover objetos
- **Stacks:** El ultimo elemento en entrar es el primero en salir (**LIFO**)
- **Colas:** El primer elemento en entrar es el primero en salir (**FIFO**)
- **Diccionarios:** guarda información de la forma llave-valor. Permiten acceso rápido por clave. Muy útiles para organizar datos por identificadores. El diccionario permite utilizar como llaves los siguientes objetos inmutables: `int`, `float`, `str`, `bool`, `tuple` (siempre que todos sus elementos también sean inmutables)
- **Sets:** funcionan como conjuntos matemáticos y es muy eficiente para determinar si un elemento esta o no en el conjunto
- **Namedtuples:** Similar a una tupla, pero permite acceder a los valores por nombre de atributo (`namedtupla.valor`) mejorando la legibilidad.

***args y **kwargs**

- `*args`**:** Se usa para pasar un número variable de argumentos posicionales a una función. Los valores se reciben como una tupla.
- `**kwargs`**:** Se usa para pasar un número variable de argumentos por clave a una función. Los valores se reciben como un diccionario.
- **Uso combinado:** Es posible combinar ambos para mayor flexibilidad en la definición de funciones, respetando el orden: `*args` primero y `**kwargs` después.
- **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 esencial en la programación orientada a objetos que permite que una clase (*subclase* o *clase hija*) herede atributos y métodos de otra clase (*superclase* o *clase padre*). Esto facilita la reutilización del código, la organización jerárquica y la extensión de funcionalidades de manera modular.
- **Multiherencia:** Python permite que una clase herede de múltiples clases base, lo que puede ser útil en casos donde se necesita combinar características de distintas clases. En este contexto, Python utiliza el algoritmo de **MRO (Method Resolution Order)** para decidir el orden en que se buscan los métodos o atributos en las clases superiores, siguiendo una jerarquía definida de izquierda a derecha en la declaración de herencia.
- `super()`: es una función clave en entornos de herencia múltiple, ya que permite acceder a métodos y atributos de clases base respetando el orden de la MRO. Esto garantiza que todas las clases involucradas sean correctamente inicializadas o sus métodos sean llamados en el orden esperado.
- **Clases Abstractas:** Las clases abstractas son estructuras que sirven como plantilla para otras clases. Estas clases no están destinadas a ser instanciadas directamente, sino que actúan como modelos que definen métodos obligatorios (*abstractos*) que deben ser implementados por las subclases. Esto fomenta la consistencia en la implementación de características comunes entre diferentes clases derivadas.
- **Properties:** Las `properties` son una herramienta que permite definir métodos que actúan como atributos, ofreciendo un control detallado sobre el acceso y modificación de datos encapsulados en una clase. Esto es útil para garantizar integridad, realizar validaciones o implementar lógica adicional al interactuar con los atributos. Existen tres componentes principales:
  - **Getter:** Permite acceder al valor de un atributo, ofreciendo la posibilidad de calcularlo dinámicamente o de acceder a un atributo privado de manera controlada.
  - **Setter:** Proporciona control adicional al asignar valores a un atributo, permitiendo implementar validaciones, restricciones o transformaciones en el proceso.
  - **Deleter:** Facilita la adición de comportamiento personalizado al eliminar un atributo.

1.3. Iterables e Iteradores en Python

Iterables

Un **iterable** es un objeto que contiene una colección de elementos y puede ser recorrido uno a uno mediante un ciclo `for`. Estos objetos deben implementar el método especial `__iter__()`, que retorna un **iterador**.

- **Ejemplos:** listas, tuplas, sets, diccionarios.
- Los iterables pueden recorrerse múltiples veces.

Iteradores

Un **iterador** es un objeto que permite recorrer un iterable. Debe implementar los métodos especiales:

- `__iter__()`: Retorna el mismo iterador (`self`).
- `__next__()`: Devuelve el siguiente elemento del iterable. Si no quedan elementos, levanta una excepción `StopIteration`.

Los iteradores **se consumen**: no pueden reiniciarse, pero es posible crear nuevos iteradores a partir del iterable original.

Clases Iterables e Iteradoras

Podemos hacer que una clase sea iterable implementando los métodos `__iter__()` y `__next__()`. Esto es útil para personalizar el comportamiento de iteración.

```
class MiIterable:
    def __init__(self, datos: list):
        self.datos = datos

    def __iter__(self):
        return MiIterador(self.datos)


class MiIterador:
    def __init__(self, datos):
        self.datos = datos.copy()

    def __iter__(self):
        return self

    def __next__(self):
        if not self.datos:
            raise StopIteration
        return self.datos.pop(0)
```

Generadores

Los **generadores** son iteradores que generan elementos sobre la marcha, sin almacenarlos en memoria. Se crean con una función que utiliza la palabra clave `yield`.

Ventajas

- **Eficiencia de memoria:** No almacenan todos los datos en memoria, generándolos solo cuando se necesitan.
- **Más legibles** que implementar una clase iteradora desde cero.

1.4. Listas Ligadas

Estructura Nodal

Un nodo es la unidad base de las estructuras de datos como listas ligadas. Cada nodo contiene:

- Un **valor**: La información almacenada en el nodo.
- Una referencia al **nodo siguiente** (también llamado sucesor). En caso de ser el último nodo de la lista debe ser `None`, esto con el objetivo de terminar la iteración.

Lista Ligada

Una lista ligada es una estructura secuencial que almacena nodos conectados mediante referencias. Se define por dos nodos clave:

- **Cabeza (head):** Es el primer nodo de la lista.
- **Cola (tail):** Es el último nodo de la lista, cuya referencia al sucesor es `None`.

Operaciones Principales

Agregar (`add`): Este método permite insertar un nuevo nodo al final de la lista ligada. Los pasos generales son:

1. Crear un nuevo nodo con el valor proporcionado.
2. Actualizar la referencia del nodo `cola` actual para apuntar al nuevo nodo.

3. Actualizar el atributo `cola` de la lista para que apunte al nuevo nodo.

En caso de que la lista esté vacía, el nuevo nodo se convierte en `cabeza` y `cola`.

Obtener (`get`): Este método recupera el valor almacenado en el nodo que se encuentra en una posición específica de la lista:

- Se recorre la lista desde la cabeza hasta alcanzar la posición deseada.
- Si la posición es mayor al largo de la lista o negativa, el método retorna un valor nulo (`None`.

Insertar (`insert`): Este método permite agregar un nuevo nodo con un valor en una posición específica de la lista ligada. Los pasos generales son:

1. Crear un nuevo nodo con el valor proporcionado.
2. Si la posición es 0, se inserta como nueva `cabeza`, actualizando las referencias de manera correspondiente.
3. Si la posición es mayor al largo de la lista, el método no realiza la inserción.
4. Para posiciones intermedias:
   - Se localiza el nodo anterior a la posición deseada.
   - Se actualizan las referencias del nodo anterior para apuntar al nuevo nodo.
   - El nuevo nodo apunta al sucesor original del nodo anterior.

Casos Borde de Inserción:

- **Lista Vacía:** Si la lista está vacía y se inserta en la posición 0, el nuevo nodo será la `cabeza` y la `cola`.
- **Posición 0:** El nuevo nodo reemplaza a la `cabeza` actual y su sucesor apunta al nodo que previamente era la `cabeza`.
- **Posición Mayor al Largo:** La inserción no se realiza y se notifica al usuario que la posición es inválida.

1.5. Programación Funcional

Es un paradigma de programación que se basa en el uso de funciones puras (lo ideal) para manipular datos y estructuras. Sus principales objetivos son:

- **Inmutabilidad:** Los datos no se modifican, sino que se crean nuevos conjuntos de datos cuando se realizan operaciones.
- **Funciones Puras:** Se busca minimizar los efectos secundarios. Una función pura siempre devuelve el mismo resultado dado un mismo conjunto de parámetros.
- **Reutilización y Composición:** Permite combinar funciones simples para crear operaciones más complejas.

La programación funcional se utiliza ampliamente en análisis de datos, procesamiento paralelo y desarrollo de sistemas que requieren un manejo complejo de datos o una alta modularidad. Este paradigma facilita el manejo de iterables y la transformación eficiente de datos.

Funciones

Funciones Lambda

- Son funciones anónimas, es decir, no tienen un nombre explícito. Se utilizan cuando necesitamos funciones simples que no se reutilizan en otras partes del programa.
- Sintaxis: `lambda argumentos: expresión`.
- Ejemplo:

```
cuadrado = lambda x, y: x * y
```

`map`

- Aplica una función a cada elemento de uno o más iterables y devuelve un iterable con los resultados.
- Ideal para transformar datos en bloque.
- Ejemplo:

```
numeros = [1, 2, 3, 4]
cuadrados = map(lambda x: x ** 2, numeros)
print(list(cuadrados))  # Salida: [1, 4, 9, 16]
```

`filter`

- Filtra los elementos de un iterable basado en una función que devuelve `True` o `False`.
- Se utiliza para seleccionar elementos que cumplen con una condición específica.
- Ejemplo:

```
numeros = [1, 2, 3, 4, 5, 6]
pares = filter(lambda x: x % 2 == 0, numeros)
print(list(pares))  # Salida: [2, 4, 6]
```

`reduce`

- Busca reducir un iterable a un único valor, aplicando una función acumulativa.
- Requiere importar `reduce` desde el módulo `functools`.
- Ejemplo:

```
from functools import reduce

numeros = [1, 2, 3, 4]
suma = reduce(lambda x, y: x + y, numeros)
print(suma)  # Salida: 10
```

1.6. Excepciones

Las excepciones en Python permiten manejar errores de manera sistemática, brindando la posibilidad de personalizar su comportamiento. Al ser una clase, las excepciones pueden heredarse y adaptarse a necesidades específicas, lo que las hace sumamente versátiles.

Manejo de Excepciones

- **`raise`:** Se utiliza para levantar manualmente excepciones, interrumpiendo el flujo del programa si se cumple una condición específica. Se pueden emplear excepciones predefinidas en Python, como `ValueError` o `TypeError`. Esto es útil para garantizar que el programa no continúe en un estado no válido.

```
def verificar_9(numeros):
    if numeros[0] != "9":
        raise ValueError("Formato equivocado")
    print("No llegare aqui!")
```

- **`try/except`:** Permite manejar excepciones de forma controlada. El bloque `try` intenta ejecutar un conjunto de instrucciones. Si ocurre una excepción, el bloque `except` la captura y ejecuta un código alternativo. Esto asegura que el programa no se detenga abruptamente.

```
try:
    nombre = "Juan Perez"
    numero_asociado = int(nombre)
except ValueError:
    print("Error: no se puede convertir el nombre en un numero.")
```

Excepciones comunes en Python

- `ValueError`: Ocurre cuando una función recibe un argumento con un tipo de dato correcto, pero con un valor inapropiado (por ejemplo, convertir un string a int).
- `TypeError`: Se lanza cuando se intenta realizar una operación no válida entre tipos incompatibles (por ejemplo, sumar un entero con una string).
- `IndexError`: Se genera al intentar acceder a un índice fuera del rango de una lista o tupla.
- `KeyError`: Ocurre cuando se intenta acceder a una llave inexistente en un diccionario.
- `ZeroDivisionError`: Se lanza al intentar dividir un número entre cero.

2. Primer Bloque - Ayudantía Presencial

2.1. Threading

Conceptos Básicos

- **Thread:** Un `Thread` es un objeto que permite ejecutar un trozo de código de manera paralela al programa `main`. Para utilizarlo, primero debemos definir una función objetivo y luego inicializarlo con el método `thread.start()`. Un thread puede terminar su ejecución de manera independiente al programa principal, y múltiples threads pueden correr de forma concurrente.
- **Daemon Thread:** Un `Daemon Thread` es un thread que se ejecuta en segundo plano y no bloquea la finalización del programa principal. Para convertir un thread en `Daemon`, se utiliza el método `setDaemon(True)` antes de llamarlo con `start()`.
  - Si el programa principal termina, todos los `Daemon Threads` se detienen automáticamente.
  - Es importante evitar operaciones críticas en `Daemon Threads`, ya que no garantizan completar su ejecución si el programa principal finaliza.
- **Timer:** Un `Timer` es un tipo específico de `Thread` que ejecuta un proceso solo una vez después de un tiempo establecido. Esto resulta útil para operaciones programadas que no necesitan ser recurrentes.

Sincronización de Threads

Para coordinar y sincronizar múltiples threads en un programa, se pueden utilizar los siguientes métodos:

- `join()`: Este método detiene la ejecución de un hilo (principal o secundario) hasta que el thread especificado haya finalizado su ejecución. Es útil para garantizar que un proceso paralelo se complete antes de continuar con la ejecución.
- **Locks:** Los `Locks` son herramientas que permiten bloquear el acceso a recursos compartidos entre threads. Con un lock, se asegura que solo un thread tenga acceso exclusivo a información delicada en un momento dado. Es importante liberar el lock con `release()` después de utilizarlo para evitar bloqueos generales en el programa
- **Events:** Los `Events` son mecanismos que permiten sincronizar múltiples threads. Se pueden imaginar como un semáforo que controla cuándo un thread puede continuar. Los métodos principales son:
  - `set()`: Da la "luz verde" a los threads para avanzar.
  - `clear()`: Pone la "luz roja", deteniendo los threads.
  - `wait()`: Bloquea el thread actual hasta que reciba la "luz verde".
  - `is_set`: Devuelve un valor booleano indicando si la "luz verde" está activa (`True`) o si está detenida (`False`).

Comparación de Métodos de Sincronización

Método	Función Principal	Ventajas	Limitaciones
`join()`	Detiene el hilo de ejecución hasta que el thread especifico finalice.	Sencillo de implementar y controlar.	Bloquea la ejecución completa, lo que puede ser ineficiente si hay múltiples threads.
Locks	Controla el acceso exclusivo a recursos compartidos para evitar conflictos entre threads.	Evita colisiones y garantiza la consistencia de datos en recursos compartidos.	Puede causar "deadlocks" si no se libera correctamente, complicando el flujo del programa.
Eventos (Events)	Sincroniza threads mediante señales como `set()`, `clear()` y `wait()`.	Permite una sincronización más flexible y dinámica entre múltiples threads.	Puede volverse complejo en programas con demasiados threads o señales simultáneas, dificultando el "debugep".

Cuadro 1: Comparación de Métodos de Sincronización en `threading`

2.2. Bytes y Serialización

Manejo de Bytes

Un byte es una secuencia de 8 bits, donde cada bit puede ser un 0 o un 1. Esto permite representar $2^8 = 256$ combinaciones únicas. En Python, los bytes se utilizan para manejar datos binarios, y se representan mediante dos tipos principales de objetos:

- `bytes`**:** Objeto inmutable, similar a un string. Se utiliza para almacenar secuencias de datos binarios que no necesitan ser modificadas.
- `bytearray`**:** Objeto mutable, similar a una lista. Permite modificar los datos almacenados en la secuencia de bytes.

El uso de `encode()` y `decode()` permite convertir entre cadenas de texto y objetos de bytes:

- `"texto".encode(‘‘utf-8’’)`: Convierte un string en bytes.
- `b’bytes’.decode(‘‘utf-8’’)`: Convierte un objeto de bytes en un string utilizando el encoding especificado.

Serialización

La serialización consiste en convertir un objeto en una representación de bytes que luego puede ser deserializada para reconstruir el objeto original. Las dos herramientas principales para serialización son **JSON** y **pickle**, cada una con características específicas:

JSON (JavaScript Object Notation)

- Serializa tipos básicos de datos como `int`, `str`, `float`, `dict`, `bool`, `list`, `tuple` y `NoneType`.
- Es un estándar legible por humanos y soportado por múltiples lenguajes de programación.
- La serialización genera un string, que luego puede ser convertido a bytes si es necesario (`.encode()`).
- Limitaciones: No permite serializar objetos personalizados ni estructuras más complejas de Python.

Pickle

- Permite serializar cualquier objeto de Python, incluidos objetos personalizados, clases, funciones y estructuras complejas.
- La serialización genera directamente un objeto de bytes, no legible para humanos.
- Es específico de Python y no puede ser utilizado en otros lenguajes.
- Riesgos de seguridad: Puede ejecutar código malicioso al deserializar datos no confiables.

Comparación entre JSON y Pickle

Aspecto	JSON	Pickle
Compatibilidad	Multilenguaje	Solo Python
Legibilidad	Legible para humanos	No legible (bytes)
Tipos soportados	Tipos básicos (`int`, `str`, etc.)	Cualquier objeto de Python
Seguridad	Seguro	Riesgo de ejecución de código malicioso

Cuadro 2: Comparación entre JSON y Pickle

Networking

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.

Arquitectura Cliente-Servidor

La arquitectura cliente-servidor consiste en una interacción donde un computador ofrece servicios y espera conexiones de clientes, estos realizan solicitudes y el servidor responde según el protocolo establecido (TCP o UDP).

Sockets

Un *socket* es un objeto del sistema operativo, que permite la comunicación entre programas, ya sea en máquinas diferentes o en la misma máquina (usando distintos puertos):

- Los **sockets de servidor** utilizan el método `bind((host, port))` para asociarse a una dirección y puerto, y `listen()` para esperar conexiones.
- Los **sockets de cliente** se conectan al servidor usando `connect((host, port))`.

Manejo de Múltiples Conexiones

Un servidor puede manejar múltiples conexiones usando **threads**. Para esto, tiene un thread que esta atento escuchando nuevas solicitudes de conexión. Ademas, cada vez que un cliente se conecta, el servidor crea un nuevo thread que maneja exclusivamente la comunicación con ese cliente. Esto permite atender a varios clientes de manera concurrente sin bloquear el programa principal.

Cliente Conectado a Múltiples Servidores

Un cliente puede conectarse a múltiples servidores al mismo tiempo mediante la creación de un `socket` independiente por servidor. Cada conexión utiliza un thread separado para enviar y recibir datos de manera concurrente.

Arquitectura Peer-to-Peer (P2P)

En la arquitectura P2P, no hay un servidor central. Cada nodo de la red actúa simultáneamente como cliente y servidor. Para lograr esto, cada nodo necesita dos `sockets`: uno para aceptar conexiones entrantes (servidor) y otro para conectarse a otros nodos (cliente). Esta estructura distribuye la carga entre todos los nodos, eliminando puntos únicos de fallo.

3. Segundo Bloque - Ayudantía Presencial

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.

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.

La consulta: 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

3.0.1. 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 códigos están organizados en familias según su propósito. Los más frecuentes son:

- **1XX (Informativo):** Indican que la solicitud fue recibida y se está procesando, pero no contienen la respuesta final del servidor. Ejemplo:
  - **100 Continue:** El cliente puede continuar con su solicitud.
- **2XX (Éxito):** Todo salió bien, la solicitud fue exitosa. Ejemplo:
  - **200 OK:** Respuesta estándar para una solicitud exitosa.
  - **201 Created:** Indica que un recurso fue creado exitosamente.
- **3XX (Redirección):** Indican que la solicitud debe redirigirse a otra ubicación para completarse. Ejemplo:
  - **301 Moved Permanently:** El recurso solicitado ha sido movido permanentemente a otra URL.
  - **302 Found:** El recurso se encuentra en otra ubicación temporalmente.
- **4XX (Errores del Cliente):** El cliente se equivocó en su solicitud. Ejemplo:
  - **400 Bad Request:** La solicitud no es válida o tiene un error de sintaxis.
  - **401 Unauthorized:** El cliente no proporcionó credenciales válidas.
  - **404 Not Found:** La ruta o recurso solicitado no existe.
- **5XX (Errores del Servidor):** Algo falló en el servidor al procesar la solicitud. Ejemplo:
  - **500 Internal Server Error:** Un error inesperado ocurrió en el servidor.
  - **503 Service Unavailable:** El servidor no está disponible temporalmente, por ejemplo, debido a mantenimiento.

Headers: Siguen la misma lógica de los headers del cliente. Es decir exponen meta-información.

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

Servidor Básico con Flask

El siguiente código muestra cómo implementar un servidor básico utilizando Flask en Python, incluyendo los métodos HTTP `GET`, `POST` y `PATCH`:

```
from flask import Flask, request, jsonify

# Crear la aplicacion Flask
app = Flask(__name__)

# Definir una ruta GET
@app.route('/')
def home():
    return 'Bienvenido al servidor Flask!'

# Definir una ruta POST
@app.route('/crear', methods=['POST'])
def crear():
    data = request.json
    return jsonify({"mensaje": f"Datos recibidos: {data}"}), 201

# Definir una ruta PATCH
@app.route('/actualizar', methods=['PATCH'])
def actualizar():
    data = request.json
```

```
    return jsonify({"mensaje": f"Datos actualizados: {data}"}), 200

# Ejecutar el servidor
if __name__ == '__main__':
    app.run(debug=True)
```

Explicación del Código

- **Creación de la aplicación:** `app = Flask(__name__)` crea una instancia de la clase `Flask`, que representa nuestra aplicación web.
- **Definición de rutas con decoradores:** Los decoradores como `@app.route` permiten asociar rutas específicas con funciones que procesan las solicitudes.
  - `GET`: La ruta `/` está asociada al método `GET` por defecto. Cuando un cliente accede a esta ruta, se ejecuta la función `home()`, devolviendo un mensaje de bienvenida.
  - `POST`: La ruta `/crear` está configurada para aceptar solicitudes `POST`. La función `crear()` usa `request.json` para obtener el cuerpo de la solicitud en formato JSON y devuelve una respuesta con código de estado `201 Created`.
  - `PATCH`: La ruta `/actualizar` acepta solicitudes `PATCH`. Similar a `POST`, la función `actualizar()` procesa datos JSON enviados por el cliente y devuelve una respuesta con código de estado `200 OK`.
- **Método `request`:** El objeto `request` permite acceder a los datos enviados por el cliente. En este caso, se usa `request.json` para procesar datos en formato JSON.

3.1. PyQt: Interfaces Gráficas y Concurrencia

Acoplamiento y Cohesión

- **Cohesión:** Se refiere a qué tan bien las tareas realizadas por un módulo están relacionadas entre sí. Alta cohesión significa que un módulo se enfoca en una sola responsabilidad, facilitando la comprensión y el mantenimiento.
- **Acoplamiento:** Describe el grado de interdependencia entre módulos. Bajo acoplamiento implica que los módulos interactúan de manera mínima, permitiendo que los cambios en uno afecten lo menos posible a los demás.

Ejecución Lineal vs Arquitectura Basada en Eventos

- **Ejecución Lineal:** Las tareas se ejecutan secuencialmente, una tras otra. Esto puede ser ineficiente en sistemas interactivos, donde una operación bloqueante (como esperar un clic) detendría todo el programa.
- **Arquitectura Basada en Eventos:** Las tareas se disparan en respuesta a **eventos** (como un clic o una tecla presionada). Esto permite ejecutar múltiples acciones de forma concurrente y no bloqueante, lo que es ideal para interfaces gráficas.

Eventos Comunes en PyQt

- `button.clicked()`: Se activa cuando se hace clic en un botón.
- `mousePressEvent`: Detecta clics del ratón.
- `keyPressEvent`: Captura la pulsación de teclas en un componente gráfico.

Separación Frontend-Backend

La separación entre **Frontend** y **Backend** es esencial para construir sistemas robustos y escalables:

- **Frontend (Front):** Maneja todo lo relacionado con la interfaz gráfica y la interacción con el usuario. Por ejemplo, botones, cuadros de texto, y ventanas.

- **Backend (Back):** Implementa la lógica de negocio y las funcionalidades que procesan las acciones realizadas por el usuario en el Frontend.

Esta separación facilita un **bajo acoplamiento** (los cambios en el Frontend afectan poco al Backend y viceversa) y una **alta cohesión** (cada componente se centra en su propia tarea). Un ejemplo de comunicación back-front sería:

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.

QThreads

Los **QThreads** permiten manejar tareas concurrentes sin bloquear la interfaz gráfica. A diferencia de los `threading.Thread` tradicionales, los `QThreads` están diseñados para integrarse mejor con PyQt, permitiendo comunicación mediante señales. Sin embargo no es posible darles una función target con argumento a los QThreads, por lo cual estos tipicamente se heredan y se sobrescribe su método `.run()`

3.2. Tópicos Avanzados en Python

Expresiones Regulares (Regex)

- **Función principal:** Las expresiones regulares permiten buscar, identificar, y manipular patrones de texto dentro de cadenas de caracteres.
- **Cuándo utilizarlas:** Son útiles para tareas como:
  - Validar entradas de usuario (por ejemplo, correos electrónicos o RUTs).
  - Extraer información específica de textos (como fechas, precios o citas).
  - Reemplazar patrones dentro de un texto (como eliminar caracteres especiales).
- **Creación de un patrón:** Para crear patrones en Python, se utiliza el módulo `re`. Ejemplo de un patrón para buscar palabras que comiencen con "a":

```
import re
patron = r'\ba\w*'
```

- **Métodos importantes:**
  - `re.match`: Comprueba si el patrón coincide al inicio del texto.
  - `re.fullmatch`: Verifica si el texto completo coincide con el patrón.
  - `re.search`: Busca el patrón en cualquier parte del texto.
  - `re.sub`: Reemplaza las coincidencias del patrón por un nuevo valor.
  - `re.split`: Divide el texto en partes utilizando el patrón como separador.

Grafos

- **Definición:** Un grafo es una estructura compuesta por nodos (o vértices) conectados mediante aristas. Es útil para modelar relaciones entre entidades, como redes sociales, mapas de rutas, o redes de networking.
- **Tipos de grafos:**
  - **Dirigidos:** Las conexiones entre nodos tienen una dirección (A $\rightarrow$ B).
  - **No dirigidos:** Las conexiones no tienen dirección (A $\leftrightarrow$ B).

- **Búsqueda en grafos:** Existen diversas formas para recorrer un grafo y saber por ejemplo si existe un determinado valor dentro de un grafo, o si es posible encontrar un camino entre 2 nodos. El curso se centra principalmente en:
  - **BFS (Breadth-First Search):** Explora los nodos por niveles de cercanía, particularmente útil para encontrar la distancia más corta en grafos. Utiliza una cola para guardar los nodos pendientes de explorar
  - **DFS (Depth-First Search):** Explora lo más profundo posible en un camino antes de retroceder. Utiliza una pila o stack para guardar los nodos pendientes de explorar

  Es importante mencionar que tanto BFS como DFS recorren en su totalidad el grafo

3.2.1. Numpy y Pandas

Numpy

- **Qué es:** Numpy es una librería para computación numérica que permite trabajar eficientemente con arrays multidimensionales y realizar operaciones matemáticas complejas.
- **Arrays de Numpy:** Son estructuras optimizadas para almacenar y manipular datos numéricos. A diferencia de las listas, los arrays son más rápidos y consumen menos memoria.
- **Ventajas en matemáticas:** Numpy permite realizar operaciones vectorizadas sobre matrices y vectores, lo que reduce la necesidad de bucles y mejora el rendimiento. Ejemplo:

```
import numpy as np
matriz = np.array([[1, 2], [3, 4]])
transpuesta = matriz.T
```

Pandas

- **Qué es:** Pandas es una librería para análisis y manipulación de datos que se basa en estructuras como **dataframes**.
- **Dataframe:** Es una estructura bidimensional similar a una tabla, que permite organizar y analizar datos tabulares de manera eficiente.
- **Ventajas:** Pandas facilita la limpieza, transformación, y análisis de datos gracias a operaciones como filtrado, agrupamiento y manejo de valores faltantes. Ejemplo:

```
import pandas as pd
data = {'Nombre': ['Ana', 'Juan'], 'Edad': [25, 30]}
df = pd.DataFrame(data)
print(df)
```