Python: APIs

1. ¿Qué es una API?

    - Definición de API (Application Programming Interface).

    Una API (Application Programming Interface) es un conjunto de reglas y protocolos que permiten que diferentes aplicaciones se comuniquen entre sí. Es una interfaz de software que define cómo interactuar con un sistema o servicio específico. En esencia, una API establece la forma en que los componentes de software deben comunicarse y cómo acceder a determinadas funcionalidades o datos.

    Una API es un conjunto de métodos, funciones, clases y protocolos que ofrecen una biblioteca, un sistema operativo o cualquier otro servicio para ser utilizado por otros software. Proporciona una capa de abstracción que permite a los desarrolladores acceder y utilizar las funcionalidades de un sistema o servicio sin tener que conocer los detalles internos de su implementación.

    Una API define los métodos disponibles, los parámetros que se deben proporcionar, los formatos de datos admitidos y las respuestas que se pueden esperar. En resumen, la API establece una interfaz bien definida que facilita la interacción entre diferentes componentes de software.

    - Importancia y uso de las API en el desarrollo de software:

    Las API desempeñan un papel fundamental en el desarrollo de software y ofrecen varios beneficios:

    Reutilización de código: Las API permiten que diferentes aplicaciones utilicen y compartan funcionalidades comunes sin necesidad de desarrollarlas desde cero. Esto promueve la reutilización de código y acelera el proceso de desarrollo.

    Modularidad: Las API permiten que las aplicaciones se dividan en componentes independientes y se comuniquen entre sí de manera estandarizada. Esto facilita la gestión y el mantenimiento del software, ya que los cambios realizados en una API no afectarán directamente a otras partes del sistema.

    Interoperabilidad: Las API facilitan la integración entre diferentes sistemas y aplicaciones, incluso si están desarrolladas en diferentes lenguajes de

programación o se ejecutan en plataformas distintas. Esto permite que los sistemas existentes se conecten entre sí y compartan información de manera eficiente.

Facilidad de actualización: Al utilizar una API, los desarrolladores pueden realizar cambios internos en una aplicación sin afectar a los usuarios o a otras aplicaciones que dependen de esa API. Esto proporciona una mayor flexibilidad para realizar actualizaciones y mejoras sin interrupciones significativas.

Acceso a servicios externos: Las API permiten a las aplicaciones acceder a servicios y datos proporcionados por terceros. Por ejemplo, las API de servicios de mapas, redes sociales, sistemas de pago, entre otros, brindan a los desarrolladores la capacidad de incorporar funcionalidades adicionales en sus aplicaciones sin tener que desarrollarlas desde cero.

En resumen, las API son fundamentales en el desarrollo de software moderno, ya que facilitan la comunicación, la reutilización de código, la integración y la interoperabilidad entre diferentes sistemas y aplicaciones.

2. Tipos de API:

    a. Tipos de API:

        - API web: Las API basadas en la web son aquellas que utilizan los protocolos de Internet para permitir la comunicación y el intercambio de datos entre diferentes aplicaciones a través de la web. Algunos de los tipos más comunes de API web son:

            - REST (Representational State Transfer): Es un estilo arquitectónico que define un conjunto de principios y restricciones para el diseño de servicios web. Las API RESTful se basan en los métodos HTTP (GET, POST, PUT, DELETE) para realizar operaciones sobre recursos que se identifican mediante URLs. Utilizan formatos de datos como JSON o XML para representar la información.

            - SOAP (Simple Object Access Protocol): Es un protocolo de intercambio de mensajes en formato XML utilizado para acceder a servicios web. Las API basadas en SOAP definen operaciones específicas y su formato de datos utilizando el lenguaje de descripción de servicios web (WSDL). SOAP utiliza el protocolo HTTP, pero también puede funcionar sobre otros protocolos como SMTP o TCP.

        - GraphQL: Es un lenguaje de consulta y una especificación para la comunicación entre cliente y servidor en aplicaciones web. Permite que el cliente especifique exactamente los datos que necesita y reduce el ancho de banda requerido al enviar solo la información solicitada. GraphQL ofrece una gran flexibilidad al cliente para obtener los datos necesarios y evitar el problema de las sobre o subconsultas.

    - API de bibliotecas: Además de las API web, también existen API que se utilizan para interactuar con bibliotecas o frameworks específicos. Estas API proporcionan un conjunto de funciones y métodos predefinidos que permiten a los desarrolladores interactuar con la funcionalidad y los componentes proporcionados por la biblioteca. Algunos ejemplos de API de bibliotecas son:
        - API de Python para manipulación de bases de datos: Por ejemplo, la API de SQLite en Python proporciona métodos para conectarse, ejecutar consultas y manipular datos en una base de datos SQLite.
        - API de TensorFlow: Es una biblioteca de aprendizaje automático (machine learning) muy popular. La API de TensorFlow ofrece una interfaz para construir y entrenar modelos de aprendizaje automático.
        - API de Django: Es un framework de desarrollo web en Python. La API de Django proporciona una interfaz para construir aplicaciones web de manera eficiente utilizando el patrón Modelo-Vista-Controlador (MVC).

3. Protocolos de comunicación:

    - HTTP (Hypertext Transfer Protocol):

    El HTTP (Hypertext Transfer Protocol) es un protocolo de comunicación utilizado en la mayoría de las API web y es la base de la World Wide Web. Funciona como un protocolo de solicitud-respuesta entre un cliente y un servidor.

    Cuando un cliente realiza una solicitud HTTP a un servidor, se establece una conexión y se envía una solicitud que consta de un método (como GET, POST, PUT, DELETE), una URL y posiblemente algunos encabezados y datos adicionales. El servidor procesa la solicitud y devuelve una respuesta al cliente, que incluye un código de estado HTTP y los datos solicitados.

    - Métodos HTTP comunes:

Los métodos HTTP son verbos o acciones que indican qué tipo de operación se debe realizar en un recurso en el servidor. Algunos de los métodos HTTP más comunes son:

1. GET: Se utiliza para recuperar información de un recurso en el servidor. La solicitud GET no debe tener un efecto secundario en los datos del servidor y solo se utiliza para leer información.
2. POST: Se utiliza para enviar datos al servidor para crear un nuevo recurso. La solicitud POST puede tener un efecto secundario en los datos del servidor, como agregar un nuevo registro a una base de datos.
3. PUT: Se utiliza para enviar datos al servidor para actualizar un recurso existente. La solicitud PUT reemplaza completamente el recurso existente con los datos proporcionados.
4. DELETE: Se utiliza para eliminar un recurso en el servidor. La solicitud DELETE elimina el recurso especificado.

Además de estos, existen otros métodos como PATCH (para realizar actualizaciones parciales en un recurso) y OPTIONS (para obtener información sobre las opciones de comunicación disponibles).

- Códigos de estado HTTP:

Los códigos de estado HTTP son números de tres dígitos que se incluyen en las respuestas HTTP para indicar el estado del resultado de la solicitud.

Descripción general de los códigos de error más comunes:

1. Códigos de estado en la serie 100: Estos códigos de estado son informativos y no suelen encontrarse con frecuencia en las respuestas de las API:
    - 100 (Continue): Indica que el servidor ha recibido los encabezados iniciales de la solicitud y que el cliente puede continuar enviando el cuerpo de la solicitud.
2. Códigos de estado en la serie 200: Estos códigos de estado indican que la solicitud se ha procesado correctamente:
    - 200 (OK): Indica que la solicitud ha tenido éxito y que la respuesta contiene los datos solicitados. Es el código de estado más común para respuestas exitosas.

- 201 (Created): Indica que la solicitud ha sido procesada correctamente y que se ha creado un nuevo recurso como resultado.
- 204 (No Content): Indica que la solicitud se ha procesado correctamente, pero la respuesta no contiene ningún contenido. Se utiliza comúnmente en solicitudes de eliminación exitosas.

3. Códigos de estado en la serie 300: Estos códigos de estado indican redirecciones:
    - 301 (Moved Permanently): Indica que el recurso solicitado ha sido movido permanentemente a una nueva ubicación. El cliente debe actualizar sus referencias al recurso utilizando la nueva URL proporcionada en la respuesta.
    - 302 (Found) / 307 (Temporary Redirect): Indica que el recurso solicitado ha sido temporalmente movido a una nueva ubicación. El cliente debe redirigir la solicitud a la nueva URL proporcionada en la respuesta.
4. Códigos de estado en la serie 400: Estos códigos de estado indican errores en la solicitud realizada por el cliente:
    - 400 (Bad Request): Indica que la solicitud no se pudo entender o procesar debido a un formato incorrecto o información faltante. Puede ser causado por parámetros inválidos o una estructura JSON incorrecta, por ejemplo.
    - 401 (Unauthorized): Indica que se requiere autenticación para acceder al recurso solicitado. El cliente debe proporcionar credenciales válidas en la solicitud.
    - 403 (Forbidden): Indica que el servidor ha entendido la solicitud, pero se niega a autorizarla. El cliente no tiene permiso para acceder al recurso solicitado.
    - 404 (Not Found): Indica que el recurso solicitado no ha sido encontrado en el servidor.
5. Códigos de estado en la serie 500: Estos códigos de estado indican errores en el servidor:
    - 500 (Internal Server Error): Indica un error interno del servidor. Es un código de estado genérico utilizado cuando el servidor encuentra una

condición inesperada que le impide procesar la solicitud correctamente.

- 502 (Bad Gateway): Indica que el servidor actuó como intermediario y recibió una respuesta no válida del servidor ascendente al intentar cumplir la solicitud.
- 503 (Service Unavailable): Indica que el servidor no está disponible temporalmente para manejar la solicitud debido a una sobrecarga o mantenimiento del servidor.

4. Introducción a la biblioteca `requests` en Python:
   La biblioteca `requests` es una biblioteca de Python que facilita realizar solicitudes HTTP de manera sencilla. Proporciona una interfaz simple y legible que permite enviar solicitudes y recibir respuestas de servidores web.

   Con `requests`, puedes enviar solicitudes HTTP utilizando diferentes métodos (GET, POST, PUT, DELETE, etc.), agregar encabezados personalizados, enviar datos en el cuerpo de la solicitud y manejar fácilmente las respuestas recibidas.

   La biblioteca `requests` abstrae gran parte de la complejidad de las comunicaciones HTTP subyacentes y proporciona una interfaz de alto nivel que facilita el uso de API web y la interacción con servicios basados en la web.

   - Instalación de la biblioteca y configuración del entorno de desarrollo:
     Para comenzar a utilizar la biblioteca `requests`, debes asegurarte de tenerla instalada en tu entorno de desarrollo. Puedes instalarla fácilmente utilizando `pip`, el administrador de paquetes de Python.

   1. Abre una terminal o línea de comandos.
   2. Ejecuta el siguiente comando para instalar la biblioteca `requests`:

      ```
      pip install requests
      ```

   Una vez que la instalación se complete, estás listo para empezar a utilizar `requests` en tu código Python.

   Para utilizar la biblioteca `requests`, simplemente importa el módulo `requests` en tu script de Python:

```
import requests
```

A partir de este punto, puedes utilizar las funciones y métodos proporcionados por `requests` para realizar solicitudes HTTP y manejar las respuestas recibidas.

Aquí tienes un ejemplo práctico que utiliza `requests` para realizar una solicitud GET a una API pública y mostrar el contenido de la respuesta:

```
import requests

# Realizar una solicitud GET a una API pública
response = requests.get('https://api.example.com/users')

# Verificar el código de estado de la respuesta
if response.status_code == 200:
    # Mostrar el contenido de la respuesta
    print(response.text)
else:
    print('Error en la solicitud:', response.status_code)
```

En este ejemplo, se realiza una solicitud GET a la URL `'https://api.example.com/data'`. Luego, se verifica el código de estado de la respuesta para asegurarse de que la solicitud fue exitosa (código de estado 200). Si es exitosa, se muestra el contenido de la respuesta utilizando `response.text`. Si ocurre algún error, se imprime el código de estado de la respuesta.

Ejemplo: API GITHUB

Aquí veremos cómo utilizar `requests` para realizar una solicitud GET a la API de GitHub para obtener información sobre un usuario:

```
import requests

# Definir el token de acceso personal (personal access token)
token = 'tu_token_de_acceso_personal'

# Especificar el encabezado de autenticación
headers = {'Authorization': f'token {token}'}
```

```
# Realizar una solicitud GET a la API de GitHub para obtener información del usuario
response = requests.get('https://api.github.com/users/tu_usuario_de_github', headers=
headers)

# Verificar el código de estado de la respuesta
if response.status_code == 200:
    # Mostrar el contenido de la respuesta
    user_data = response.json()
    print('Nombre de usuario:', user_data['login'])
    print('Nombre completo:', user_data['name'])
    print('Ubicación:', user_data['location'])
else:
    print('Error en la solicitud:', response.status_code)
```

En este ejemplo, debes reemplazar `'tu_token_de_acceso_personal'` con tu propio token de acceso personal de GitHub, y `'tu_usuario_de_github'` con tu nombre de usuario de GitHub. La solicitud GET se realiza a la URL `'https://api.github.com/users/tu_usuario_de_github'` para obtener información del usuario. Luego, se muestra parte de la información recibida en la respuesta.

5. Realización de solicitudes con `requests`:
   - Creación de solicitudes GET, POST, PUT y DELETE utilizando `requests`:
     La biblioteca `requests` proporciona funciones para crear solicitudes HTTP utilizando diferentes métodos. Aquí tienes ejemplos de cómo utilizar `requests` para realizar solicitudes GET, POST, PUT y DELETE:

     1. Solicitud GET:

```
import requests

response = requests.get('https://api.example.com/data')
```

     2. Solicitud POST:

```
import requests

data = {'key': 'value'}
response = requests.post('https://api.example.com/endpoint', data=data)
```

3. Solicitud PUT:

```
import requests

data = {'key': 'value'}
response = requests.put('https://api.example.com/endpoint', data=data)
```

4. Solicitud DELETE:

```
import requests

response = requests.delete('https://api.example.com/endpoint')
```

En cada ejemplo, se utiliza la función correspondiente de `requests` (`get()`, `post()`, `put()`, `delete()`) y se proporciona la URL del endpoint al que se desea enviar la solicitud.

- Inclusión de parámetros en las solicitudes (query parameters):
  Los parámetros de consulta (query parameters) se utilizan para agregar información adicional a una URL. Pueden ser utilizados para filtrar resultados, especificar orden, paginación, etc. Puedes incluir parámetros de consulta en tus solicitudes utilizando el parámetro `params` de `requests`. Aquí tienes un ejemplo:

```
import requests

params = {'param1': 'value1', 'param2': 'value2'}
response = requests.get('https://api.example.com/endpoint', params=params)
```

En este ejemplo, `params` es un diccionario que contiene los parámetros de consulta que deseas incluir en la solicitud GET. `requests` los agrega automáticamente a la URL.

- Envío de datos en el cuerpo de las solicitudes (request body):
  En algunas solicitudes, como POST y PUT, puedes enviar datos en el cuerpo de la solicitud. Puedes hacerlo pasando los datos al parámetro `data` de `requests`. Aquí tienes un ejemplo:

```
import requests

data = {'key1': 'value1', 'key2': 'value2'}
response = requests.post('https://api.example.com/endpoint', data=data)
```

En este ejemplo, `data` es un diccionario con los datos que deseas enviar en el cuerpo de la solicitud POST. `requests` se encarga de codificarlos y enviarlos adecuadamente.

- Manejo de encabezados (headers) en las solicitudes:
  Puedes especificar encabezados personalizados en tus solicitudes utilizando el parámetro `headers` de `requests`. Los encabezados se utilizan para enviar información adicional en la solicitud, como información de autenticación, tipos de contenido aceptados, etc. Aquí tienes un ejemplo:

```
import requests

headers = {'Content-Type': 'application/json', 'Authorization': 'Bearer your_token'}
response = requests.get('https://api.example.com/endpoint', headers=headers)
```

En este ejemplo, `headers` es un diccionario que contiene los encabezados personalizados que deseas incluir en la solicitud. Puedes especificar múltiples encabezados separándolos por comas.

Métodos más utilizados:

1. GET (Obtener):
   El método GET se utiliza para recuperar información de un recurso en el servidor. Es el método más comúnmente utilizado y se emplea para solicitar datos del servidor sin realizar cambios en el estado de los recursos.

Cuando se realiza una solicitud GET, los parámetros de la solicitud, como las consultas de búsqueda, se incluyen en la URL. El servidor procesa la solicitud y devuelve una respuesta con los datos solicitados, generalmente en formato JSON, XML, HTML u otro formato especificado en la respuesta.

Por ejemplo, al realizar una solicitud GET a la siguiente URL, se obtendrían los detalles de un usuario con el ID 123:

```
https://api.example.com/users/123
```

1. POST (Enviar):
   El método POST se utiliza para enviar datos al servidor para crear un nuevo recurso. Se emplea para enviar información adicional en el cuerpo de la solicitud, que el servidor utilizará para realizar las acciones necesarias, como almacenar los datos en una base de datos.

Cuando se realiza una solicitud POST, los datos se envían en el cuerpo de la solicitud en un formato específico, como JSON o formularios codificados. La respuesta del servidor puede incluir información sobre el nuevo recurso creado, como un código de estado 201 (Creado) y la ubicación del recurso.

Por ejemplo, al realizar una solicitud POST a la siguiente URL, se crearía un nuevo usuario con la información proporcionada en el cuerpo de la solicitud:

```
https://api.example.com/users

Datos en el cuerpo de la solicitud:
{
  "nombre": "Juan",
  "apellido": "Pérez",
  "correo": "juan@example.com"
}
```

1. PUT (Actualizar):
   El método PUT se utiliza para enviar datos al servidor y actualizar un recurso existente. A diferencia del método POST, donde se crea un nuevo recurso,

PUT se utiliza para reemplazar o actualizar completamente un recurso existente con los datos proporcionados en el cuerpo de la solicitud.

Al realizar una solicitud PUT, se envía la URL del recurso que se va a actualizar junto con los nuevos datos en el cuerpo de la solicitud. El servidor procesa la solicitud y realiza los cambios necesarios en el recurso.

Por ejemplo, al realizar una solicitud PUT a la siguiente URL, se actualizaría el usuario con el ID 123 con los nuevos datos proporcionados:

```
https://api.example.com/users/123

Datos en el cuerpo de la solicitud:
{
  "nombre": "Pedro",
  "apellido": "Gómez",
  "correo": "pedro@example.com"
}
```

Es importante destacar que estos son solo ejemplos y las URLs y formatos de datos pueden variar según la API específica que estés utilizando.

6. Manipulación de respuestas:

   - Recepción y manejo de respuestas HTTP:
     Después de enviar una solicitud utilizando `requests`, recibirás una respuesta HTTP del servidor. Puedes utilizar `requests` para acceder y manipular diferentes aspectos de la respuesta recibida.

   Ejemplo de cómo recibir y manejar una respuesta HTTP:

```
import requests

response = requests.get('https://api.example.com/data')

# Acceder al código de estado de la respuesta
status_code = response.status_code

# Acceder al contenido de la respuesta
content = response.text

# Acceder a los encabezados de la respuesta
```

```
headers = response.headers
```

En este ejemplo, `response` es la respuesta HTTP recibida después de realizar una solicitud GET. Puedes acceder al código de estado de la respuesta utilizando `response.status_code`, al contenido de la respuesta utilizando `response.text` y a los encabezados de la respuesta utilizando `response.headers`.

- Obtención de datos de respuesta, como el contenido de la respuesta y los encabezados:
  La biblioteca `requests` proporciona diferentes atributos y métodos para acceder a los datos de la respuesta.

Algunos de los métodos y atributos útiles son:

- `response.text`: Devuelve el contenido de la respuesta como una cadena de texto.
- `response.json()`: Devuelve el contenido de la respuesta interpretado como JSON y lo convierte en un diccionario de Python.
- `response.headers`: Devuelve un diccionario que contiene los encabezados de la respuesta.
- `response.status_code`: Devuelve el código de estado de la respuesta (por ejemplo, 200 para una respuesta exitosa).

Aquí tienes un ejemplo de cómo utilizar estos métodos:

```
import requests

response = requests.get('https://api.example.com/data')

# Obtener el contenido de la respuesta como texto
content = response.text
print('Contenido:', content)

# Obtener el contenido de la respuesta como un diccionario JSON
data = response.json()
print('Datos:', data)

# Obtener los encabezados de la respuesta
headers = response.headers
print('Encabezados:', headers)
```

```
# Obtener el código de estado de la respuesta
status_code = response.status_code
print('Código de estado:', status_code)
```

- Validación de códigos de estado y manejo de errores:
  Cuando recibes una respuesta HTTP, es importante validar el código de estado para determinar si la solicitud fue exitosa o si se produjo algún error. Los códigos de estado proporcionan información sobre el estado de la solicitud y pueden variar desde 2xx para solicitudes exitosas hasta 4xx o 5xx para errores.

Puedes utilizar los códigos de estado para tomar decisiones en tu código y manejar situaciones de error de manera apropiada. Aquí tienes un ejemplo:

```
import requests

response = requests.get('https://api.example.com/data')

if response.status_code == 200:
    # La solicitud fue exitosa
    content = response.text
    # Realizar acciones adicionales con el contenido de la respuesta
else:
    # Ocurrió un error en la solicitud
    print('Error:', response.status_code)
    # Realizar acciones para manejar el error
```

En este ejemplo, se verifica si el código de estado de la respuesta es 200 (indicando una respuesta exitosa). Si es así, se procede a trabajar con el contenido de la respuesta. De lo contrario, se imprime el código de estado y se puede implementar el manejo de errores correspondiente

7. Autenticación en API:

   Existen varios métodos de autenticación comunes utilizados en las API para garantizar la seguridad y controlar el acceso a los recursos. Algunos de los métodos de autenticación más utilizados son:

   1. API Keys (Claves de API): Las claves de API son cadenas únicas generadas por el proveedor de la API y se utilizan para autenticar las solicitudes. Se

envían como parte de la solicitud, generalmente en forma de un encabezado o un parámetro, para identificar al usuario o a la aplicación que realiza la solicitud.

2. Tokens de acceso: Los tokens de acceso son cadenas de caracteres generadas después de que un usuario se autentica correctamente. Estos tokens se utilizan para autorizar las solicitudes posteriores realizadas en nombre del usuario autenticado. Los tokens de acceso se envían generalmente en el encabezado de la solicitud, típicamente en un encabezado de autenticación.
3. OAuth (Open Authorization): OAuth es un protocolo de autorización ampliamente utilizado para permitir que una aplicación acceda a los recursos de un usuario sin revelar las credenciales de inicio de sesión del usuario. Con OAuth, un usuario puede autorizar a una aplicación a acceder a sus datos en una API específica en su nombre. Esto se logra mediante la emisión de tokens de acceso que se utilizan para autenticar las solicitudes de la aplicación en nombre del usuario.

- Cómo agregar autenticación a las solicitudes realizadas con `requests`:

La biblioteca `requests` proporciona varias formas de agregar autenticación a las solicitudes realizadas. A continuación, se muestra un ejemplo de cómo agregar autenticación utilizando diferentes métodos:

1. Autenticación con API Key:

```
import requests

headers = {'API-Key': 'your_api_key'}
response = requests.get('https://api.example.com/endpoint', headers=headers)
```

   En este ejemplo, se agrega la clave de API al encabezado de la solicitud utilizando el parámetro `headers`.

2. Autenticación con Token de Acceso:

```
import requests
```

```
headers = {'Authorization': 'Bearer your_access_token'}
response = requests.get('https://api.example.com/endpoint', headers=headers)
```

En este ejemplo, se agrega el token de acceso al encabezado de la solicitud utilizando el parámetro `headers`. El prefijo "Bearer" se utiliza para indicar que se trata de un token de autenticación.

3. Autenticación con OAuth:
   La autenticación con OAuth puede requerir un flujo más complejo, ya que implica la obtención de un token de acceso válido mediante el intercambio de credenciales y tokens temporales. La implementación exacta dependerá del proveedor de la API y del flujo de OAuth utilizado.

En general, deberás seguir los pasos proporcionados por el proveedor de la API para obtener un token de acceso válido y luego incluirlo en tus solicitudes utilizando el encabezado de autenticación, similar al método de autenticación con token de acceso mencionado anteriormente.

8. Prácticas recomendadas:

```
import requests

# Definir la URL base de la API y el endpoint
base_url = 'http://api.openweathermap.org/data/2.5/weather'
api_key = 'your_api_key'

# Definir parámetros de la solicitud
params = {
    'q': 'London,uk',  # Ciudad y código de país para obtener el clima
    'appid': api_key  # Clave de API para autenticación
}

try:
    # Realizar la solicitud GET a la API
    response = requests.get(base_url, params=params)

    # Verificar si la solicitud fue exitosa (código de estado 200)
    if response.status_code == 200:
        # Obtener los datos de la respuesta en formato JSON
        data = response.json()

        # Acceder a los datos relevantes de la respuesta
        temperature = data['main']['temp']
        weather_description = data['weather'][0]['description']
```

```
        # Mostrar los resultados
        print('Temperatura:', temperature)
        print('Descripción del clima:', weather_description)
    else:
        # Mostrar el código de estado en caso de error
        print('Error en la solicitud:', response.status_code)

except requests.exceptions.RequestException as e:
    # Capturar errores de solicitud
    print('Error de solicitud:', e)
```

En este ejemplo, se utiliza la OpenWeatherMap API para obtener datos del clima de una ciudad específica. Se siguen algunas buenas prácticas:

1. Manejo de errores: Se utiliza un bloque `try-except` para capturar y manejar posibles errores durante la solicitud. Esto ayuda a evitar que la aplicación se bloquee si ocurren errores.
2. Control de límites de tasa (rate limiting): Algunas API pueden tener restricciones sobre el número de solicitudes que se pueden realizar en un cierto período de tiempo. Para respetar estas limitaciones, es importante revisar la documentación de la API para conocer cualquier límite de tasa establecido y ajustar el código en consecuencia.
3. Uso de versiones de API: Algunas API pueden tener múltiples versiones disponibles. Es una buena práctica especificar la versión de la API que se está utilizando en la URL o en los encabezados de la solicitud, para asegurarte de que tu código funcione de manera consistente incluso si la API se actualiza en el futuro.

REST APIs:

REST (Representational State Transfer) es un estilo arquitectónico para el diseño de sistemas distribuidos basados en la web. Se utiliza ampliamente en el desarrollo de API (Application Programming Interfaces) para permitir la comunicación y transferencia de datos entre diferentes aplicaciones o servicios. A continuación, se desarrollan los aspectos clave de las REST API:

1. Principios de REST:

- Arquitectura cliente-servidor: Las REST API siguen el modelo cliente-servidor, donde el cliente (aplicación, dispositivo o servicio) realiza solicitudes a un servidor que proporciona los recursos o servicios solicitados.
- Sin estado (stateless): Cada solicitud que realiza el cliente al servidor debe contener toda la información necesaria para comprender y procesar la solicitud. El servidor no mantiene ningún estado sobre las solicitudes anteriores del cliente.
- Operaciones sobre recursos: Los recursos son entidades identificables en el sistema, como usuarios, productos o publicaciones. Las REST API utilizan los métodos HTTP (GET, POST, PUT, DELETE, etc.) para realizar operaciones sobre estos recursos.
- Interfaz uniforme: Las REST API siguen una interfaz uniforme que incluye la identificación de recursos mediante URLs (Uniform Resource Locators) y el uso de formatos estándar para la representación de datos, como JSON (JavaScript Object Notation) o XML (eXtensible Markup Language).

1. Estructura de una REST API:
    - URL base: Define la dirección base de la API, a través de la cual se accede a los recursos.
    - Recursos: Representan las entidades que se pueden acceder y manipular mediante la API. Cada recurso tiene una URL única.
    - Métodos HTTP: Se utilizan para realizar operaciones sobre los recursos. Los métodos comunes son GET (obtener), POST (crear), PUT (actualizar) y DELETE (eliminar).
    - Códigos de estado HTTP: Indican el resultado de una solicitud. Algunos ejemplos son 200 (éxito), 201 (creado), 400 (solicitud incorrecta) y 404 (recurso no encontrado).
    - Parámetros: Permiten personalizar las solicitudes mediante la inclusión de información adicional, como filtros, paginación o ordenamiento.
2. Beneficios de las REST API:
    - Flexibilidad: Las REST API permiten una comunicación flexible y escalable entre diferentes sistemas, ya que utilizan estándares web ampliamente

adoptados.

- Independencia de plataforma: Las REST API son independientes de la plataforma y el lenguaje de programación, lo que facilita la integración entre aplicaciones desarrolladas en diferentes tecnologías.
- Separación entre cliente y servidor: La arquitectura cliente-servidor de las REST API permite una clara separación de responsabilidades y promueve la modularidad y la reutilización del código.
- Alta interoperabilidad: Al utilizar protocolos web estándar como HTTP y formatos de datos comunes como JSON, las REST API son compatibles con una amplia gama de tecnologías y pueden interactuar con diferentes sistemas.

3. Buenas prácticas al diseñar una REST API:
    - Utilizar URLs semánticas y significativas para los recursos.
    - Emplear los métodos HTTP de manera adecuada y coherente según las operaciones realizadas sobre los recursos.
    - Utilizar códigos de estado HTTP apropiados para indicar el resultado de las solicitudes.

Manipulación de los datos de respuesta para extraer información relevante.