Request en Python

python

INDICE

1. Introducción a HTTP

1.1 ¿Qué es HTTP?

HTTP (HyperText Transfer Protocol) es el lenguaje que usan los programas para hablar entre sí en Internet. Es como el "idioma común" que permite que tu navegador hable con un servidor web.

Analogía del restaurante (para entender HTTP)

Imagina que HTTP funciona como un restaurante:

- Cliente (tú/tu navegador): El comensal que pide comida.
- Servidor (web): La cocina que prepara la comida.
- Request (petición): Tu pedido escrito en papel.
- Response (respuesta): El plato de comida que te traen.
- URL: La dirección del restaurante.
- Método HTTP: El tipo de acción (pedir, modificar pedido, cancelar).

Ejemplo real

Cuando escribes en el navegador: https://www.youtube.com/watch?v=abc123
1. Tu navegador (cliente) hace una REQUEST al servidor de YouTube.
2. El servidor procesa tu petición.
3. El servidor envía una RESPONSE con el video.
4. Tu navegador muestra el video.

1.2 Anatomía de una petición HTTP

Una petición HTTP tiene 4 partes fundamentales:

1. Método (GET, POST, PUT, DELETE…): Qué acción quiero hacer.
2. URL (https://api.com/posts/1): A dónde va dirigida.
3. HEADERS (metadatos):
   - Content-Type: application/json.
   - Authorization: Bearer token123.
4. BODY (contenido, opcional): {"title": "Mi post"}.

1.3 Métodos HTTP: Las 5 operaciones fundamentales

Los métodos HTTP son las acciones que puedes realizar sobre un recurso.

Método	Acción	Ejemplo Real	¿Tiene BODY?
GET	Leer/Consultar	Ver perfil de usuario	NO
POST	Crear nuevo	Registrar usuario nuevo	SI
PUT	Actualizar TODO	Editar perfil completo	SI
PATCH	Actualizar PARCIAL	Solo cambiar el email	SI
DELETE	Eliminar	Borrar cuenta	NO

Explicación detallada de cada método

- GET - Obtener información (READ): Quiero ver algo, pero no modificarlo.

```
# Ejemplo 1: Listar todos los posts
GET https://api.blog.com/posts

# Ejemplo 2: Ver un post específico
GET https://api.blog.com/posts/42

# Ejemplo 3: Buscar posts de un usuario
```

GET https://api.blog.com/posts?userId=5&limit=10

Características de GET:

- No lleva BODY (los datos van en la URL como parámetros).
- Se puede cachear (el navegador puede guardar la respuesta).
- Se puede marcar como favorito (la URL tiene toda la info)

- POST – Crear algo nuevo (CREATE): Quiero crear algo nuevo en el servidor.

```
# Ejemplo: Crear un nuevo post
POST https://api.blog.com/posts
Body: {
   "title": "Mi primer post",
   "content": "Contenido del post",
   "author": "Miguel"
}

# Respuesta típica:
Status: 201 Created
Body: {
   "id": 123,
   "title": "Mi primer post",
   "content": "Contenido del post",
   "author": "Miguel",
   "createdAt": "2025-12-15T10:30:00Z"
}
```

Características de POST:

- Lleva BODY con los datos a crear.
- No se cachea
- Retorna status 201 Created si tiene éxito

- PUT – Reemplazar completamente (UPDATE FULL): Quiero reemplazar todo el recurso.

```
# Ejemplo: Actualizar un post completo (TODOS los campos).
PUT https://api.blog.com/posts/123
Body: {
   "id": 123,
   "title": "Título actualizado",
   "content": "Contenido actualizado",
   "author": "Miguel"
}

# CUIDADO: Si no envías un campo, se puede borrar.
PUT https://api.blog.com/posts/123
Body: {
   "title": "Solo título"
   #Falta content y author - se pueden perder
}
```

Características de PUT:

- Lleva BODY con TODOS los campos.
- Es idempotente (llamarlo 10 veces = mismo resultado).

- Retorna status 200 OK o 204 No Content.

- PATCH - Actualizar parcialmente (UPDATE PARTIAL): Solo quiero cambiar ALGUNOS campos.

```
# Ejemplo: Cambiar solo el título
PATCH https://api.blog.com/posts/123
Body: {
   "title": "Nuevo título"
   #Los demás campos (content, author) NO se tocan
}

# Ejemplo: Cambiar título y contenido
PATCH https://api.blog.com/posts/123
Body: {
   "title": "Nuevo título",
   "content": "Nuevo contenido"
   #El campo author permanece igual
}
```

- PUT vs PATCH - Diferencia:

```
# Recurso original:
{
   "id": 1,
   "title": "Post original",
   "content": "Contenido original",
   "author": "Miguel",
   "tags": ["python", "web"]
}

# Con PUT (reemplaza TODO):
PUT /posts/1
Body: {"title": "Nuevo título"}
Resultado: {
   "id": 1,
   "title": "Nuevo título"
   # Se perdió: content, author, tags
}

# Con PATCH (actualiza SOLO lo enviado):
PATCH /posts/1
Body: {"title": "Nuevo título"}
Resultado: {
   "id": 1,
   "title": "Nuevo título",
   "content": "Contenido original",   # Se mantiene
   "author": "Miguel",                # Se mantiene
   "tags": ["python", "web"]          # Se mantiene
}
```

- **DELETE – Eliminar recurso:** Quiero borrar algo.

 # Ejemplo 1: Eliminar un post
DELETE https://api.blog.com/posts/123

 # Respuesta típica:
Status: 200 OK o 204 No Content
Body: {} o {"message": "Post eliminado"}

 # Ejemplo 2: Eliminar todos los posts de un usuario
DELETE https://api.blog.com/users/5/posts

Características de DELETE:

- Generalmente no lleva BODY.
- Retorna 200 OK, 204 No Content o 404 Not Found.

2. URLLIB: El método nativo de Python

2.1 ¿Qué es urllib?

urllib es el módulo incluido en Python para hacer peticiones HTTP. Viene instalado por defecto, así que no necesitas pip install.

Ventajas:

- Ya viene con Python (no necesitas instalar nada).
- Útil para scripts simples o entornos restringidos.
- Es la forma "oficial" de hacer HTTP en Python.

Desventajas:

- Código muy verboso (muchas líneas para cosas simples).
- Manejo tedioso de bytes y encoding.
- Sintaxis poco intuitiva.
- No tiene funcionalidades modernas (sesiones, retry, etc.).

Nota: urllib es útil para entender cómo funciona HTTP por debajo, pero en proyectos reales usaremos requests (que veremos en la siguiente sección).

2.2 GET simple con urllib

Vamos a hacer una petición GET a JSONPlaceholder (https://jsonplaceholder.typicode.com/), una API de prueba gratuita que simula un backend con posts, usuarios y comentarios.

```
import urllib.request
import json

# URL de la API de prueba (devuelve lista de posts)
api_url = "https://jsonplaceholder.typicode.com/posts"

try:
    # PASO 1: Abrir la URL (esto hace la petición HTTP GET)
    print("Haciendo petición GET...")
    respuesta = urllib.request.urlopen(api_url)
```

```
    # PASO 2: Leer los datos que devuelve el servidor
    # Importante: urlopen() devuelve BYTES, no texto
    datos_bytes = respuesta.read()
    print(f"Recibidos {len(datos_bytes)} bytes")

    # PASO 3: Convertir bytes a string UTF-8
    datos_texto = datos_bytes.decode("utf-8")

    # PASO 4: Convertir string JSON a diccionario Python
    posts = json.loads(datos_texto)

    # PASO 5: Usar los datos
    print(f"Total de posts recibidos: {len(posts)}")
    print(f"Primer post: '{posts[0]['title']}'")
    print(f"Autor del primer post: Usuario {posts[0]['userId']}")

    # PASO 6: Cerrar la conexión (buena práctica)
    respuesta.close()
    print("Conexión cerrada")

except urllib.error.URLError as e:
    print(f"Error al realizar la solicitud: {e}")
    print("Verifica tu conexión a internet o la URL")

except json.JSONDecodeError as e:
    print(f"Error al parsear JSON: {e}")
    print("La respuesta del servidor no es JSON válido")
```

Salida esperada:

```
Haciendo petición GET...
Recibidos 27281 bytes
Total de posts recibidos: 100
Primer post: 'sunt aut facere repellat provident occaecati excepturi optio reprehenderit'
Autor del primer post: Usuario 1
Conexión cerrada
```

Version más compacta:

```
import urllib.request
```

```python
import json

try:
    api_url = "https://jsonplaceholder.typicode.com/posts"
    respuesta = urllib.request.urlopen(api_url)
    datos = respuesta.read()
    posts = json.loads(datos.decode("utf-8"))
    respuesta.close()

    print(f"Total de posts: {len(posts)}")
    print(f"Primer post: {posts[0]['title']}")

except urllib.error.URLError as e:
    print(f"Error: {e}")
```

2.3 GET con parámetros (Query Parameters)

A veces necesitas filtrar los datos con parámetros en la URL.
Ejemplo: Obtener solo los posts del usuario 1.

```python
import urllib.request
import urllib.parse
import json

try:
    # FORMA 1: Construir URL manualmente (NO recomendado)
    api_url = "https://jsonplaceholder.typicode.com/posts?userId=1"

    # FORMA 2: Usar urlencode (RECOMENDADO)
    base_url = "https://jsonplaceholder.typicode.com/posts"
    parametros = {
        'userId': 1,
        'limit': 5
    }

    # Convertir diccionario a query string: userId=1&limit=5
    query_string = urllib.parse.urlencode(parametros)

    # Construir URL completa
    url_completa = f"{base_url}?{query_string}"
    print(f"URL generada: {url_completa}")
```

```
    # Hacer la petición
    respuesta = urllib.request.urlopen(url_completa)
    datos = respuesta.read()
    posts = json.loads(datos.decode("utf-8"))
    respuesta.close()

    print(f"Posts del usuario 1: {len(posts)}")
    for post in posts[:3]:  # Mostrar solo los primeros 3
        print(f"  - {post['title']}")

except urllib.error.URLError as e:
    print(f"Error: {e}")
```

Salida esperada:

```
URL generada: https://jsonplaceholder.typicode.com/posts?userId=1&limit=5
Posts del usuario 1: 10
- sunt aut facere repellat provident...
- qui est esse
- ea molestias quasi exercitationem...
```

2.4 POST con urllib - Enviar datos al servidor

POST es más complejo que urllib porque hay que:

- Convertir los datos a JSON.
- Convertir el JSON a bytes.
- Crear un objeto Request con headers.
- Especificar el método POST.

Ejemplo: Crear un nuevo post.

```
import urllib.request
import json

try:
    api_url = "https://jsonplaceholder.typicode.com/posts"

    # PASO 1: Preparar los datos que queremos enviar
    nuevo_post = {
        "title": "Mi nuevo post desde Python",
        "body": "Este es el contenido de mi post de prueba",
```

```
        "userId": 1
    }
    print(f"Enviando: {nuevo_post}")

    # PASO 2: Convertir diccionario Python → JSON string → bytes
    json_string = json.dumps(nuevo_post)
    json_bytes = json_string.encode('utf-8')

    # PASO 3: Crear objeto Request con método POST y headers
    request = urllib.request.Request(
        api_url,
        data=json_bytes,  # El body en bytes
        headers={
            'Content-Type': 'application/json',  # Indicar que enviamos JSON
            'Accept': 'application/json'         # Indicar que esperamos JSON
        },
        method='POST'  # Especificar el método HTTP
    )

    # PASO 4: Hacer la petición
    print("Enviando petición POST...")
    respuesta = urllib.request.urlopen(request)

    # PASO 5: Leer la respuesta del servidor
    datos_respuesta = respuesta.read()
    post_creado = json.loads(datos_respuesta.decode('utf-8'))

    # PASO 6: Verificar el resultado
    print(f"Status Code: {respuesta.status}")  # Debería ser 201
    print(f"Post creado con ID: {post_creado['id']}")
    print(f"Título: {post_creado['title']}")

    respuesta.close()

except urllib.error.HTTPError as e:
    # Errores HTTP específicos (404, 500, etc.)
    print(f"Error HTTP {e.code}: {e.reason}")

except urllib.error.URLError as e:
    # Errores de conexión
```

```
    print(f"Error de conexión: {e}")

except json.JSONDecodeError as e:
    print(f"Error al procesar JSON: {e}")
```

Salida esperada:
```
Enviando: {'title': 'Mi nuevo post desde Python', 'body': 'Este es el contenido de mi post de prueba',
'userId': 1}
Enviando petición POST...
Status Code: 201
Post creado con ID: 101
Título: Mi nuevo post desde Python
```

2.5 PUT con urllib - Actualizar un recurso completo

```
import urllib.request
import json

try:
    # Actualizar el post con ID 1
    post_id = 1
    api_url = f"https://jsonplaceholder.typicode.com/posts/{post_id}"

    # Datos actualizados (TODOS los campos)
    post_actualizado = {
        "id": post_id,
        "title": "Título completamente actualizado",
        "body": "Cuerpo completamente actualizado",
        "userId": 1
    }

    # Convertir a JSON bytes
    json_bytes = json.dumps(post_actualizado).encode('utf-8')

    # Crear Request con método PUT
    request = urllib.request.Request(
        api_url,
        data=json_bytes,
        headers={'Content-Type': 'application/json'},
```

```
        method='PUT'
    )

    print(f"Actualizando post {post_id}...")
    respuesta = urllib.request.urlopen(request)

    resultado = json.loads(respuesta.read().decode('utf-8'))
    print(f"Post actualizado: {resultado['title']}")

    respuesta.close()

except urllib.error.HTTPError as e:
    print(f"Error HTTP {e.code}: {e.reason}")
```

2.6 DELETE CON urllib – Eliminar un recurso

```
import urllib.request

try:
    post_id = 1
    api_url = f"https://jsonplaceholder.typicode.com/posts/{post_id}"

    # DELETE no necesita body, solo el método
    request = urllib.request.Request(
        api_url,
        method='DELETE'
    )

    print(f"Eliminando post {post_id}...")
    respuesta = urllib.request.urlopen(request)

    print(f"Status: {respuesta.status}")  # Debería ser 200
    print("Post eliminado exitosamente")

    respuesta.close()

except urllib.error.HTTPError as e:
    if e.code == 404:
        print(f"Post {post_id} no existe")
    else:
        print(f"Error {e.code}: {e.reason}")
```

2.7 Agregar headers personalizados

A veces las APIs requieren headers específicos (autenticación, user-agent, etc.).

```
import urllib.request
import json

try:
    api_url = "https://jsonplaceholder.typicode.com/posts"

    # Headers personalizados
    headers = {
        'Content-Type': 'application/json',
        'User-Agent': 'MiApp-Python/1.0',
        'Accept-Language': 'es-ES',
        'Authorization': 'Bearer mi_token_secreto_123'  # Ejemplo de auth
    }

    request = urllib.request.Request(
        api_url,
        headers=headers
    )

    respuesta = urllib.request.urlopen(request)

    # Ver los headers de la respuesta
    print("Headers de la respuesta:")
    for header, valor in respuesta.headers.items():
        print(f"  {header}: {valor}")

    respuesta.close()

except urllib.error.URLError as e:
    print(f"Error: {e}")
```

2.8 Problema de urllib

Veamos un ejemplo comparativo del mismo código:
Con urllib (más de 15 líneas):

```
import urllib.request
import json

try:
```

```
    api_url = "https://jsonplaceholder.typicode.com/posts/1"
    respuesta = urllib.request.urlopen(api_url)
    datos = respuesta.read()
    texto = datos.decode('utf-8')
    post = json.loads(texto)
    respuesta.close()
    print(post['title'])
except urllib.error.URLError as e:
    print(f"Error: {e}")
```

Con request(3 líneas):

```
import requests


response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
print(response.json()['title'])
```

3. Requests – La forma profesional

3.1 ¿Por qué requests es el estándar de la industria?

requests es una librería externa creada por Kenneth Reitz que simplifica enormemente las peticiones HTTP. Es el estándar de facto en Python profesional.
Pero, qué es requests y por qué usarlo: Imagina que estás aprendiendo a conducir. urllib sería como conducir un coche con cambio manual, embrague, y teniendo que calcular mentalmente cada cambio de marcha. requests sería como conducir un coche automático moderno: tú te concentras en conducir y el coche se encarga de los detalles técnicos.

El problema con urllib

Cuando usamos urllib, tenemos que hacer manualmente muchas cosas:

- Convertir datos a bytes: Siempre tenemos que usar .encode('utf-8').
- Decodificar respuestas: Siempre tenemos que usar .decode('utf-8') y json.loads().
- Construir requests complejos: Crear objetos Request manualmente.
- Manejar headers: Sintaxis poco intuitiva con diccionarios anidados.

Ventajas de requests:

- Código ultra simple y legible.
- Manejo automático de JSON.
- Sesiones con cookies automáticas.
- Autenticación incorporada.
- Timeout fácil de configurar.
- Headers intuitivos.
- Manejo de errores específico.
- Streaming de archivos grandes.
- Usado por millones de desarrolladores.

Desventaja

- Necesita instalarlo (no viene con Python).

3.2 Instalación de requests

Importante: NUNCA instales librerías directamente en el sistema. Siempre usa entornos virtuales.

Antes de instalar requests, necesitamos entender un concepto fundamental: los entornos virtuales.
Un entorno virtual es como tener una instalación completamente separada de Python para cada proyecto. Es como tener cajas diferentes para guardar las herramientas de cada trabajo.
Problema de trabajar sin entornos virtuales:

- Si tengo un proyecto que usa requests con la versión 2.25.0 y para empezar un proyecto nuevo necesito request 2.31.0, al actualizar se pueden romper proyectos viejos que antes funcionaban correctamente.
- También puede darse el caso de que diferentes proyectos necesiten versiones diferentes, pero solo puedo tener una versión instalada.
- Otro problema es que el sistema esté lleno de librerías de todos los proyectos utilizados, generando un caos total.

Con entornos virtuales todo tiene su propia "caja de herramientas" sin afectar a los demás.

Proceso completo de instalación paso a paso

Vamos a crear un proyecto nuevo y configurarlo correctamente desde cero.

PASO 1: Crear carpeta del proyecto

Abre tu terminal y ejecuta:

```
# Crear carpeta para el proyecto
mkdir mi_proyecto_requests
cd mi_proyecto_requests
```

Ahora estás dentro de la carpeta vacía de tu proyecto.

PASO 2: Crear el entorno virtual

Ejecuta este comando:

```
python -m venv venv
```

Desglosemos qué significa esto:
- python: Llama al intérprete de Python.
- "-m venv": Ejecuta el módulo venv (virtual environment).
- "venv": El nombre de la carpeta que se creará (puede ser cualquier nombre, pero "venv" es el estándar)

Después de ejecutar esto, se crea una carpeta "venv" dentro de la carpeta del proyecto, esta carpeta contiene una copia aislada de Python solo para este proyecto.

PASO 3: Activar el entorno virtual

Ahora necesitamos "entrar" a ese entorno virtual. El comando es diferente según tu sistema operativo:
- En Windows:
  venv\Scripts\activate
- En Linux/Mac
  source venv/bin/activate

Cuando el entorno está activado, verás (venv) al principio de tu línea de comandos.
Ese (venv) te indica que estás trabajando dentro del entorno virtual.

PASO 4: Instalar requests en el entorno virtual

Ahora que estamos dentro del entorno virtual, instalamos requests:

```
pip install requests
```
Esto instala requests SOLO en este entorno virtual, no en tu sistema principal.

PASO 5: Verificar la instalación
Puedes verificar qué librerías están instaladas en tu entorno:
```
pip list
```

PASO 6: Desactivar el entorno (cuando termines de trabajar)
Cuando termines de trabajar en tu proyecto, puedes salir del entorno virtual:
```
deactivate
```

Importante: Nunca subas la carpeta venv/ a Git. En su lugar, crea un archivo requirements.txt.

Ejecuta:
```
pip freeze > requirements.txt
```

Este archivo guarda la lista de todas las librerías instaladas. Otros desarrolladores pueden recrear tu entorno con:
```
pip install -r requirements.txt
```

3.3 Tu primera petición con requests

Ahora que tenemos requests instalado, vamos a hacer nuestra primera petición. Empezaremos con lo más simple y luego añadiremos complejidad.
Concepto: GET básico
Una petición GET es la más simple de todas. Es como pedirle al servidor "dame información". No enviamos datos, solo pedimos.
Vamos a pedir la lista de posts de JSONPlaceholder.
```
import requests


response = requests.get("https://jsonplaceholder.typicode.com/posts")
posts = response.json()
print(len(posts))
```
Sí, así de simple. Tres líneas. Pero vamos a entender QUÉ está pasando en cada línea.

Línea 1: import requests
Importamos la librería. Esto hace que todas las funciones de requests estén disponibles en nuestro código.
Línea 2: response = requests.get("URL")
Esta línea hace MUCHO trabajo detrás de escena:

- Abre una conexión TCP con el servidor.
- Envía una petición HTTP GET.
- Espera la respuesta del servidor.
- Lee la respuesta completa.
- Cierra la conexión.
- Guarda todo en el objeto response.

Todo esto que con urllib nos tomaría 10+ líneas, aquí es una sola línea.

Es aconsejable poner un parámetro “timeout” al requests.get (en segundos), para que no se quede esperando indefinidamente una respuesta: requests.get(url, timeout=10).
Línea 3: posts = response.json()
El objeto response tiene varios atributos y métodos útiles. El método .json() hace esto automáticamente:

- Toma el contenido de la respuesta (que es texto).
- Lo interpreta como JSON.
- Lo convierte a estructuras de Python (listas, diccionarios).
- Nos lo devuelve listo para usar.

El objeto Response: tu ventana a la respuesta
Cuando haces una petición con requests, recibes un objeto Response. Este objeto contiene toda la información sobre la respuesta del servidor. Vamos a explorarlo en detalle.
Crea un archivo llamado explorar_response.py:

```
import requests


response = requests.get("https://jsonplaceholder.typicode.com/posts/1")
```

Ahora tenemos un objeto response. Vamos a ver toda la información que contiene.
Atributo 1: status_code
El código de estado HTTP (200, 404, 500, etc.)

```
print(f"Status code: {response.status_code}")
```

Este es un número que indica si la petición fue exitosa o no. Los códigos más comunes son:

- 200: Todo bien.
- 404: No encontrado.
- 500: Error del servidor.

Atributo 2: ok
Un booleano que te dice si la petición fue exitosa (status 200-299).

```
if response.ok:
    print("La petición fue exitosa")
else:
    print("Hubo un problema")
```

Esto es mucho mejor que escribir if response.status_code >= 200 and response.status_code < 300.

Atributo 3: headers
Los headers (metadatos) que el servidor envió en la respuesta. Es un diccionario.

```
print(f"Tipo de contenido: {response.headers['Content-Type']}")
print(f"Tamaño del contenido: {response.headers['Content-Length']}")
print(f"Servidor: {response.headers.get('Server', 'No especificado')}")
```

Los headers contienen información útil como el tipo de contenido, codificación, cookies, etc.

Atributo 4: url
La URL final de la petición (útil si hubo redirecciones).

```
print(f"URL accedida: {response.url}")
```

Atributo 5: elapsed
Un objeto timedelta que indica cuánto tardó la petición.

```
print(f"Tiempo de respuesta: {response.elapsed.total_seconds()} segundos")
```

Esto es útil para monitorear el rendimiento de las APIs.

Atributo 6: request
Información sobre la petición que enviaste. Sí, puedes ver los detalles de TU petición.

```
print(f"Método usado: {response.request.method}")
print(f"URL solicitada: {response.request.url}")
print(f"Headers enviados: {response.request.headers}")
```

Ahora vamos a los métodos para obtener el contenido de diferentes formas.

Método 1: .json()
Convierte automáticamente la respuesta JSON a objetos Python.

```
datos = response.json()
```

Detrás de escena hace: json.loads(response.text).
Este es el método que más usarás cuando trabajes con APIs REST modernas que devuelven JSON.
Método 2: .text
El contenido como string (texto).

```
contenido_texto = response.text
```

Útil cuando la respuesta no es JSON sino HTML, XML, o texto plano.
Método 3: .content
El contenido como bytes (datos binarios)

```
contenido_bytes = response.content
```

Útil cuando descargas archivos binarios como imágenes, PDFs, videos, etc.

Guardemos todo lo aprendido en un script de práctica
Crea un archivo primera_peticion.py con este código:

```
import requests


def hacer_peticion_simple():
    """
    Función que hace una petición GET simple y muestra los resultados
    """
    print("Iniciando petición a JSONPlaceholder...")

    url = "https://jsonplaceholder.typicode.com/posts"
    response = requests.get(url)

    print(f"Estado de la petición: {response.status_code}")
```

```
    if response.ok:
        posts = response.json()
        print(f"\nPetición exitosa")
        print(f"Se recibieron {len(posts)} posts")
        print(f"Tiempo de respuesta: {response.elapsed.total_seconds():.2f}s")


        print("\nPrimeros 3 posts:")
        for i, post in enumerate(posts[:3], 1):
            print(f"\n{i}. {post['title']}")
            print(f"   Usuario: {post['userId']}")
            print(f"   Contenido: {post['body'][:50]}...")
    else:
        print(f"\n✗ Error en la petición: {response.status_code}")


if __name__ == "__main__":
    hacer_peticion_simple()
```

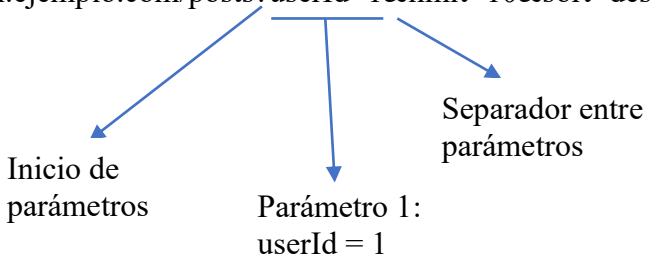
Ejecuta este script y deberías ver la información de los posts de la API.

3.4 GET con parámetros - Query Parameters

¿Qué son los Query Parameters?

Cuando navegas por internet, seguro que has visto URLs como estas:
https://www.youtube.com/watch?v=dQw4w9WgXcQ
https://www.google.com/search?q=python+requests&lang=es
https://www.amazon.es/s?k=laptop&price=100-500
La parte que viene después del símbolo ? son los query parameters (parámetros de consulta). Son como "filtros" o "especificaciones" que le das al servidor para que te devuelva exactamente lo que necesitas.
Vamos a desglosar una URL completa:

https://api.ejemplo.com/posts?userId=1&limit=10&sort=desc



Reglas de sintaxis:

1. Los parámetros empiezan después del símbolo ?
2. Cada parámetro tiene el formato: nombre=valor
3. Si hay varios parámetros, se separan con &
4. Caracteres especiales se codifican (espacios → %20, etc.)

¿Para qué sirven los Query Parameters?

Los query parameters se usan principalmente para filtrar, buscar y paginar datos:

Ejemplo 1: Filtrar posts de un usuario específico

```
GET /posts?userId=5
→ Dame solo los posts del usuario con ID 5
```

Ejemplo 2: Buscar productos

```
GET /productos?categoria=laptops&precio_max=1000&ordenar=precio_asc
→ Dame laptops que cuesten menos de 1000€, ordenados por precio ascendente
```

Ejemplo 3: Paginación

```
GET /posts?page=2&limit=20
→ Dame la página 2, mostrando 20 posts por página
```

Forma incorrecta: Construir la URL manualmente

Cuando empiezas a programar, la tentación es construir la URL manualmente concatenando strings:

```python
import requests


# FORMA INCORRECTA (pero funciona)
user_id = 1
url = "https://jsonplaceholder.typicode.com/posts?userId=" + str(user_id)
response = requests.get(url)
```

Aunque funciona hay varios problemas:

- No funciona si se usan caracteres especiales en los parámetros, como &.
- Tener múltiples parámetros es tedioso de construir,
- Problemas con valores opcionales.

Forma correcta: Usar el parámetro params

Requests tiene un parámetro especial llamado params que acepta un diccionario. Requests automáticamente:

- Codifica caracteres especiales.
- Construye la URL correctamente.
- Maneja valores None.
- Convierte números a strings.

Sintaxis básica:

```python
import requests


parametros = {
    'clave1': 'valor1',
    'clave2': 'valor2'
}


response = requests.get('https://api.ejemplo.com/endpoint', params=parametros)
```

Requests convierte automáticamente el diccionario a: ?clave1=valor1&clave2=valor2

Ejemplo 1: Filtrar posts por usuario
Vamos a obtener todos los posts de un usuario específico de JSONPlaceholder.
Primero, entendamos la API:

- Endpoint: https://jsonplaceholder.typicode.com/posts
- Sin parámetros: devuelve TODOS los posts (100 posts).
- Con ?userId=1: devuelve solo los posts del usuario 1 (10 posts).

Código sin parámetros:

```
import requests


url = "https://jsonplaceholder.typicode.com/posts"
response = requests.get(url)
posts = response.json()


print(f"Total de posts sin filtrar: {len(posts)}")
```

Esto devuelve 100 posts (de todos los usuarios).

Código con parámetros:

```
import requests


url = "https://jsonplaceholder.typicode.com/posts"


parametros = {
    'userId': 1
}


response = requests.get(url, params=parametros)
posts = response.json()


print(f"Total de posts del usuario 1: {len(posts)}")
print(f"URL final construida: {response.url}")


# Salida:
# Total de posts del usuario 1: 10
# URL final construida: https://jsonplaceholder.typicode.com/posts?userId=1
```

Fíjate que con response.url puedes ver la URL final que construyó requests. Esto es muy útil para debugging.

Ejemplo 2: Múltiples parámetros
Ahora vamos a usar varios parámetros a la vez. Imagina que quieres:

- Posts del usuario 1

- Ordenados por ID descendente
- Solo los primeros 3 resultados

JSONPlaceholder soporta estos parámetros especiales:

- _sort: campo por el cual ordenar.
- _order: asc (ascendente) o desc (descendente).
- _limit: número máximo de resultados.
- _page: número de página a mostrar (paginación).

```python
import requests

url = "https://jsonplaceholder.typicode.com/posts"

parametros = {
    'userId': 1,
    '_sort': 'id',
    '_order': 'desc',
    '_limit': 3
}

response = requests.get(url, params=parametros)
posts = response.json()

print(f"URL construida: {response.url}")
print(f"\nPrimeros 3 posts del usuario 1 (orden descendente):")
print('='*60)

for post in posts:
    print(f"ID: {post['id']} | Título: {post['title'][:40]}...")


# Salida
# URL construida: https://jsonplaceholder.typicode.com/posts?userId=1&_sort=id&_order=desc&_limit=3

# Primeros 3 posts del usuario 1 (orden descendente):
# ============================================================
# ID: 10 | Título: optio molestias id quia eum...
# ID: 9 | Título: nesciunt iure omnis dolorem tempora et...
# ID: 8 | Título: dolorem dolore est ipsam...
```

Como ves, requests construyó automáticamente:
?userId=1&_sort=id&_order=desc&_limit=3

Ejemplo 3: Parámetros opcionales

En la vida real, muchas veces los filtros son opcionales. Por ejemplo, una función de búsqueda donde el usuario puede o no especificar filtros.

Veamos cómo manejar esto elegantemente:

```
import requests


def buscar_posts(user_id=None, limite=None, ordenar_por=None):
    """
    Busca posts con filtros opcionales

    Args:
        user_id: ID del usuario (opcional)
        limite: Número máximo de resultados (opcional)
        ordenar_por: Campo para ordenar (opcional)
    """
    url = "https://jsonplaceholder.typicode.com/posts"

    # Construir diccionario de parámetros
    parametros = {}

    if user_id is not None:
        parametros['userId'] = user_id

    if limite is not None:
        parametros['_limit'] = limite

    if ordenar_por is not None:
        parametros['_sort'] = ordenar_por

    print(f"Parámetros enviados: {parametros}")

    response = requests.get(url, params=parametros)
    print(f"URL generada: {response.url}")

    return response.json()


# Prueba 1: Solo user_id
print("\n--- Búsqueda 1: Solo user_id ---")
posts1 = buscar_posts(user_id=2)
```

```python
print(f"Resultados: {len(posts1)} posts")

# Prueba 2: user_id y límite
print("\n--- Búsqueda 2: user_id + límite ---")
posts2 = buscar_posts(user_id=2, limite=5)
print(f"Resultados: {len(posts2)} posts")

# Prueba 3: Todos los parámetros
print("\n--- Búsqueda 3: Todos los parámetros ---")
posts3 = buscar_posts(user_id=1, limite=3, ordenar_por='id')
print(f"Resultados: {len(posts3)} posts")

# Prueba 4: Sin parámetros (todos los posts)
print("\n--- Búsqueda 4: Sin filtros ---")
posts4 = buscar_posts()
print(f"Resultados: {len(posts4)} posts")


# Salida
# --- Búsqueda 1: Solo user_id ---
# Parámetros enviados: {'userId': 2}
# URL generada: https://jsonplaceholder.typicode.com/posts?userId=2
# Resultados: 10 posts

# --- Búsqueda 2: user_id + límite ---
# Parámetros enviados: {'userId': 2, '_limit': 5}
# URL generada: https://jsonplaceholder.typicode.com/posts?userId=2&_limit=5
# Resultados: 5 posts

# --- Búsqueda 3: Todos los parámetros ---
# Parámetros enviados: {'userId': 1, '_limit': 3, '_sort': 'id'}
# URL generada: https://jsonplaceholder.typicode.com/posts?userId=1&_limit=3&_sort=id
# Resultados: 3 posts

# --- Búsqueda 4: Sin filtros ---
# Parámetros enviados: {}
# URL generada: https://jsonplaceholder.typicode.com/posts
# Resultados: 100 posts
```

Ejemplo 4: Caracteres especiales - Codificación automática

Uno de los grandes beneficios de usar params es que requests codifica automáticamente caracteres especiales.

```python
import requests

# Búsqueda con espacios, acentos y caracteres especiales
parametros = {
    'q': 'Python & Web Development',
    'ciudad': 'São Paulo',
    'email': 'usuario@ejemplo.com'
}

response = requests.get('https://httpbin.org/get', params=parametros)

print("Parámetros originales:")
print(parametros)
print("\nURL generada por requests:")
print(response.url)
print("\nRequests codificó automáticamente:")
print("  - Espacios → %20")
print("  - & → %26")
print("  - @ → %40")
print("  - ã → %C3%A3")



# Salida
# Parámetros originales:
# {'q': 'Python & Web Development', 'ciudad': 'São Paulo', 'email': 'usuario@ejemplo.com'}

# URL generada por requests:
#
https://httpbin.org/get?q=Python+%26+Web+Development&ciudad=S%C3%A3o+Paulo&email=usuario%4
0ejemplo.com

# Requests codificó automáticamente:
#   - Espacios → %20 (o +)
#   - & → %26
#   - @ → %40
#   - ã → %C3%A3
```

Si hubieras construido la URL manualmente, tendrías que hacer esta codificación tú mismo para tener en cuenta los caracteres especiales. Requests lo hace automáticamente.

Ejercicio 1 - ejemplo real: API del clima (OpenWeatherMap)
Vamos a ver un caso real de uso de parámetros con una API profesional. OpenWeatherMap es una API muy popular para obtener datos del clima.
Nota: Para usar OpenWeatherMap necesitas registrarte (gratis) y obtener una API key en: https://openweathermap.org/api

Crea una función llamada obtener_clima con parámetros: ciudad, api_key, unidades (metric por defecto) e idioma (es por defecto). Llama a la API de OpenWeatherMap, busca como se hace esto pulsando en la barra de menús de la web donde pone "API" y busca "Current Weather Data" y le das a "API doc". Busca para qué vale que a la respuesta de: respuesta = requests.get(…) llamemos a respuesta.raise_for_status()
Mostrar esta respuesta:

Clima en Zafra, ES
==========
Temperatura: 10.77°C
Sensación térmica: 9.91°C
Descripción: nubes
Humedad: 77%
Viento: 2.19 m/s

Ejercicio2: Buscador de posts
Crea una función llamada "menu_busqueda" que pida un ID de usuario de 1 a 10 y un límite de resultados. Crea una función main() que pida ¿Hacer otra búsqueda?(s/n) y si pulsa "s", volverá a llamar a "menu_busqueda"
Llamar a la URL: https://jsonplaceholder.typicode.com/posts y muestre los posts encontrados:

Buscando posts con filtros: {'userId': 3, '_limit': 15}
URL generada:
https://jsonplaceholder.typicode.com/posts?userId=3&_limit=15
Status: 200

Se encontraron 10 posts:

1. [Usuario 3] asperiores ea ipsam voluptatibus modi minima quia sint
repellat aliquid praesentium dolorem quo
sed totam minus non itaque
nihil labore...

2. [Usuario 3] dolor sint quo a velit explicabo quia nam
eos qui et ipsum ipsam suscipit aut
sed omnis non odio
expedita earum mollitia m...

3.5 POST con requests - Crear recursos en el servidor

¿Qué es una petición POST?

Hasta ahora con GET solo leíamos datos del servidor. Con POST vamos a enviar datos al servidor para crear algo nuevo.

Analogía del restaurante:

- GET: "¿Qué hay en el menú?" (solo consultar)
- POST: "Quiero pedir un plato de paella" (crear un nuevo pedido)

¿Cuándo usamos POST?

Usamos POST cuando queremos crear algo nuevo en el servidor:

- Registrar un nuevo usuario.
- Publicar un post en un blog.
- Subir una foto.
- Enviar un formulario de contacto.
- Crear un pedido de compra.
- Añadir un producto al carrito.

Tu primer POST con requests

Vamos a crear un post nuevo en JSONPlaceholder. Esta API simula la creación de posts (no los guarda realmente, pero responde como si lo hiciera).

Paso 1: Preparar los datos

Primero decidimos qué datos queremos enviar. En este caso, un post tiene:

- title: El título del post.
- body: El contenido del post.
- userId: El ID del usuario que lo crea.

```
# Datos que queremos enviar al servidor
nuevo_post = {
    "title": "Mi primer post desde Python",
    "body": "Este es el contenido de mi post. Estoy aprendiendo requests.",
    "userId": 1
}
```

Esto es un diccionario Python normal. Requests lo convertirá a JSON automáticamente.

Paso 2: Hacer la petición POST

```
import requests


# Datos a enviar
nuevo_post = {
    "title": "Mi primer post desde Python",
    "body": "Este es el contenido de mi post. Estoy aprendiendo requests.",
    "userId": 1
}


# URL del endpoint
url = "https://jsonplaceholder.typicode.com/posts"
```

```
# Hacer petición POST
response = requests.post(url, json=nuevo_post)


print(f"Status code: {response.status_code}")
```

¿Qué hace json=nuevo_post?

Cuando usas el parámetro "json=", requests hace automáticamente:

1. Convierte el diccionario Python a formato JSON.
2. Añade el header Content-Type: application/json.
3. Codifica el JSON como bytes UTF-8.
4. Lo envía en el body de la petición.

Para verificar si fue exitoso el POST:

```
# Verificar si fue exitoso
if response.status_code == 201:  # 201 = Created
    print("Post creado exitosamente")


    # Obtener el post que devolvió el servidor
    post_creado = response.json()


    print(f"\nID asignado por el servidor: {post_creado['id']}")
    print(f"Título: {post_creado['title']}")
    print(f"Contenido: {post_creado['body']}")
    print(f"Usuario: {post_creado['userId']}")
else:
    print(f"Error: {response.status_code}")
```

Ejercicio 3 – Crear usuario con validación

Crear una función llamada crear_post que tenga como parámetros de entrada: título, contenido y user_id, crear un post usando la URL url = https://jsonplaceholder.typicode.com/posts

Utiliza timeout de 10 segundos al hacer la petición y raise_for_status().

Código para probarlo:

```
# Crear un post
    post = crear_post(
        titulo="Python requests es genial",
        contenido="Después de aprender urllib, requests es un alivio. Todo es más simple y limpio.",
        user_id=1
    )


    if post:
        print("\n" + "="*60)
        print("OPERACIÓN COMPLETADA")
        print("="*60)
```

POST con datos más complejos

A veces necesitas enviar estructuras más complejas (listas, objetos anidados, etc.)

```
import requests
import json

# Datos complejos con estructura anidada
post_complejo = {
    "title": "Tutorial de Python",
    "body": "Contenido del tutorial...",
    "userId": 1,
    "tags": ["python", "programming", "tutorial"],  # Lista
    "metadata": {                                   # Diccionario anidado
        "views": 0,
        "likes": 0,
        "published": True,
        "category": "educación"
    },
    "related_posts": [10, 15, 23]  # Lista de IDs
}

print("Datos a enviar:")
print(json.dumps(post_complejo, indent=2, ensure_ascii=False))

url = "https://httpbin.org/post"  # Esta API devuelve lo que le envías
response = requests.post(url, json=post_complejo, timeout=10)

if response.ok:
    print("\nDatos recibidos por el servidor:")
    datos_recibidos = response.json()['json']
    print(json.dumps(datos_recibidos, indent=2, ensure_ascii=False))
```

POST con headers personalizados

A veces las APIs requieren headers adicionales (autenticación, versión de API, etc.).

```
import requests

nuevo_post = {
    "title": "Post con autenticación",
    "body": "Este post requiere estar autenticado",
    "userId": 1
```

```
}

# Headers personalizados
headers = {
    'Authorization': 'Bearer mi_token_secreto_123',
    'User-Agent': 'MiApp-Python/1.0',
    'X-API-Version': 'v2'
}

response = requests.post(
    "https://httpbin.org/post",
    json=nuevo_post,
    headers=headers,
    timeout=10
)

print("Headers enviados:")
print(json.dumps(dict(response.request.headers), indent=2))

print("\nServidor recibió:")
datos = response.json()
print(f"Authorization: {datos['headers']['Authorization']}")
print(f"User-Agent: {datos['headers']['User-Agent']}")
print(f"X-Api-Version: {datos['headers']['X-Api-Version']}")
```

Importante: Cuando usas json=, requests añade automáticamente Content-Type: application/json. Si añades headers personalizados, se combinan (no se sobrescriben).

3.6 PUT y PATCH con requests - Actualizar recursos

¿Qué son PUT y PATCH?

El concepto fundamental

Hasta ahora hemos visto:

- GET: Leer datos.
- POST: Crear datos nuevos.

Ahora vamos a aprender a modificar datos que ya existen en el servidor.
Analogía del restaurante:

- GET: "¿Qué pedí?" (consultar tu pedido).
- POST: "Quiero pedir paella" (crear pedido nuevo).
- PUT: "Cambia MI PEDIDO COMPLETO a: ensalada, pollo y postre" (reemplazar todo).
- PATCH: "Solo cambia las patatas de mi pedido por arroz" (modificar una parte).

¿Por qué existen DOS métodos para actualizar?

Esta es una de las preguntas más comunes. La respuesta está en cuánto quieres actualizar:

PUT = Reemplazo COMPLETO

- Envías TODOS los campos del recurso.
- Los campos que no envíes se pueden perder o establecer a valores por defecto.
- Es como reescribir todo el documento desde cero.

PATCH = Actualización PARCIAL

- Envías SOLO los campos que quieres cambiar.
- Los campos que no envíes se mantienen sin cambios.
- Es como usar "corrector" en partes específicas de un documento.

Entendiendo PUT: Actualización completa

Ejemplo conceptual: Perfil de usuario

Imagina que tienes este perfil de usuario en el servidor:

```
# Estado actual en el servidor (ID: 1)
usuario_actual = {
    "id": 1,
    "nombre": "Miguel",
    "email": "miguel@ejemplo.com",
    "ciudad": "Madrid",
    "telefono": "600123456",
    "biografia": "Profesor de Python"
}
```

Escenario 1: Usando PUT (reemplazar todo)

```
import requests
import json

# Quiero actualizar el email
# Con PUT debo enviar TODOS los campos
usuario_actualizado = {
    "id": 1,
    "nombre": "Miguel",
    "email": "miguel.nuevo@ejemplo.com",  # ← Campo actualizado
    "ciudad": "Madrid",
    "telefono": "600123456",
    "biografia": "Profesor de Python"
}

response = requests.put(
    "https://jsonplaceholder.typicode.com/users/1",
```

```
    json=usuario_actualizado,
    timeout=10
)

print(f"Status: {response.status_code}")  # 200
print("Usuario actualizado:")
print(json.dumps(response.json(), indent=2, ensure_ascii=False))
```

¿Qué pasó?

- Enviaste TODOS los campos.
- El servidor reemplazó el usuario completo.
- Si hubieras olvidado un campo, ese campo se hubiera perdido.

Es peligroso si no envías todos los datos, ya que se pierden todos los campos que no envías, por eso PUT es peligroso.

Entendiendo PATCH: Actualización parcial

Tenemos el mismo usuario:

```
usuario_actual = {
    "id": 1,
    "nombre": "Miguel",
    "email": "miguel@ejemplo.com",
    "ciudad": "Madrid",
    "telefono": "600123456",
    "biografia": "Profesor de Python"
}
```

Si queremos cambiar solo el email:

```
import requests

# Con PATCH solo envío lo que quiero cambiar
solo_email = {
    "email": "miguel.nuevo@ejemplo.com"
}

response = requests.patch(
    "https://api.ejemplo.com/users/1",
    json=solo_email,
    timeout=10
)

# Resultado en el servidor:
# {
#   "id": 1,
```

```
#   "nombre": "Miguel",             # Se mantiene
#   "email": "miguel.nuevo@ejemplo.com",  # Actualizado
#   "ciudad": "Madrid",             # Se mantiene
#   "telefono": "600123456",        # Se mantiene
#   "biografia": "Profesor de Python" # Se mantiene
# }
```

Solo cambió el email. Todo lo demás se mantuvo intacto.

Actualización con validación previa
En aplicaciones reales, es buena práctica obtener el recurso actual antes de actualizarlo:

```
url = f"https://jsonplaceholder.typicode.com/posts/{post_id}"
cambios={
    "title": "Nuevo título",
    "body": "Nuevo contenido"
}
# PASO 1: Obtener post actual
response_get = requests.get(url, timeout=10)
response_get.raise_for_status() #Faltaría envolverlo todo en un try except
post_actual = response_get.json()

# PASO 2: Mostrar estado actual
print(json.dumps(post_actual, indent=2, ensure_ascii=False))

# PASO 3: Aplicar cambios
print(f"Cambios solicitados: {cambios}")

post_actualizado = post_actual.copy()  # Copiar el actual
post_actualizado.update(cambios)       # Aplicar cambios

print(f"Post después de cambios:")
print(json.dumps(post_actualizado, indent=2, ensure_ascii=False))

# PASO 4: Enviar PUT
response_put = requests.put(url, json=post_actualizado, timeout=10)
response_put.raise_for_status()#Faltaría envolverlo todo en un try except
resultado = response_put.json()
```

El método .update() **mezcla** un diccionario con otro, sobrescribiendo solo las claves que coinciden.
Ventaja: No tienes que hacer esto manualmente:

```
# Sin .update() (tedioso):
```

```
persona["edad"] = cambios["edad"]


# Con .update() (simple):
persona.update(cambios)
```

Actualización optimista vs pesimista

Actualización optimista (sin verificar primero)

```
import requests


# Asumes que el recurso existe y actualizas directamente
response = requests.patch(
    "https://api.ejemplo.com/posts/1",
    json={"title": "Nuevo título"}
)


# Ventaja: Más rápido (1 petición)
# Desventaja: Si no existe, falla
```

Actualización pesimista (verificar primero)

```
import requests


# Primero verificas que existe
response_get = requests.get("https://api.ejemplo.com/posts/1")


if response_get.ok:
    # Luego actualizas
    response_patch = requests.patch(
        "https://api.ejemplo.com/posts/1",
        json={"title": "Nuevo título"}
    )


# Ventaja: Más seguro
# Desventaja: Más lento (2 peticiones)
```

Recomendación: Usa actualización optimista y maneja el error 404 si ocurre.

Regla de oro:

- **Usa PATCH** el 90% del tiempo (más seguro).
- **Usa PUT** solo cuando realmente necesites reemplazar todo.

3.7 DELETE con requests - Eliminar recursos

¿Qué es una petición DELETE?

DELETE es el método HTTP para eliminar recursos del servidor. Es la operación más destructiva de todas.
Analogía del restaurante:

- GET: "¿Qué hay en el menú?" (consultar).
- POST: "Quiero pedir paella" (crear pedido).
- PUT/PATCH: "Cambia mi pedido" (modificar).
- DELETE: "Cancela mi pedido" (eliminar).

Partes importantes:

1. **Método**: DELETE.
2. **URL**: Identifica QUÉ eliminar (/posts/1 = post con ID 1).
3. **Headers** (opcional): Autenticación si es necesario.
4. **Body**: Generalmente vacío.

Ejemplo: eliminar un post

```python
import requests

# URL del recurso a eliminar
post_id = 1
url = f"https://jsonplaceholder.typicode.com/posts/{post_id}"

print("="*60)
print(f"ELIMINANDO POST {post_id}")
print("="*60)

# Hacer petición DELETE
response = requests.delete(url, timeout=10)

# Verificar resultado
print(f"\nStatus code: {response.status_code}")

if response.status_code == 200:
    print("Post eliminado exitosamente (200 OK)")
elif response.status_code == 204:
    print("Post eliminado exitosamente (204 No Content)")
elif response.status_code == 404:
    print("Post no encontrado (404 Not Found)")
else:
    print(f"Respuesta inesperada: {response.status_code}")
```

Nota: En aplicaciones reales, es buena práctica verificar que el recurso existe antes de eliminarlo.

DELETE con autenticación

En APIs reales, DELETE suele requerir autenticación (no cualquiera puede eliminar recursos):

```python
import requests

def eliminar_con_autenticacion(post_id, token):
    """
    Elimina un post con token de autenticación

    Args:
        post_id (int): ID del post
        token (str): Token de autenticación

    Returns:
        bool: True si se eliminó
    """
    url = f"https://api.ejemplo.com/posts/{post_id}"

    # Headers con token de autenticación
    headers = {
        'Authorization': f'Bearer {token}',
        'Content-Type': 'application/json'
    }

    print(f"Eliminando post {post_id} (autenticado)...")

    try:
        response = requests.delete(
            url,
            headers=headers,
            timeout=10
        )

        if response.status_code == 401:
            print("Error 401: Token inválido o expirado")
            return False

        elif response.status_code == 403:
            print("Error 403: No tienes permiso para eliminar este recurso")
            return False
```

```
        elif response.ok:
            print("Post eliminado exitosamente")
            return True

        else:
            print(f"Error {response.status_code}")
            return False

    except requests.exceptions.RequestException as e:
        print(f"Error: {e}")
        return False
# Uso
TOKEN = "tu_token_aqui_abc123"
eliminar_con_autenticacion(1, TOKEN)
```