Extração de dados e APIs

## O que é uma API (Application Programming Interface)?

Uma API, ou "Application Programming Interface", é um conjunto de regras e convenções que permite a integração de aplicativos e sistemas. Ela é usada para permitir que aplicações externas acessem dados e funcionalidades de outras aplicações de maneira controlada e segura.

Por exemplo, imagine que você deseja integrar dados de previsão do tempo em seu aplicativo. Em vez de desenvolver a lógica para coletar esses dados por conta própria, você pode usar uma API de previsão do tempo. A API fornece a você uma maneira padronizada de acessar os dados de previsão, o que torna o processo mais fácil e eficiente.

## APIs REST

API REST (Application Programming Interface Representational State Transfer) é um padrão de projeto de API que se baseia na arquitetura da Web. Ele permite que os sistemas interajam uns com os outros por meio da troca de dados via HTTP.

Os principais recursos de uma API REST são os métodos HTTP GET, POST, PUT e DELETE. Aqui está uma explicação detalhada de cada um deles:

1. GET: o método GET é usado para recuperar informações de uma fonte de dados. Ele é o método mais comumente usado para obter dados de uma API REST;
2. POST: o método POST é usado para enviar informações para uma fonte de dados. Ele é usado para criar novos recursos ou atualizar informações existentes;
3. PUT: o método PUT é usado para atualizar informações de uma fonte de dados. Ele é semelhante ao método POST, mas ele requer que você especifique o recurso que deseja atualizar; e
4. DELETE: o método DELETE é usado para excluir informações de uma fonte de dados. Ele é usado para excluir um recurso específico.

Em resumo, as APIs REST usam os métodos HTTP para permitir que os sistemas interajam uns com os outros por meio da troca de dados via HTTP. Os métodos GET, POST, PUT e DELETE são os principais recursos de uma API REST e permitem a recuperação, envio, atualização e exclusão de informações de uma fonte de dados, respectivamente.

## Exemplos Básicos

Para facilitar nosso entendimento, vamos ver alguns exemplos de como usar os métodos GET, POST, PUT e DELETE com o pacote requests do Python:

1. Método GET

import requests

url = "https://api.example.com/users"

response = requests.get(url)

if response.status\_code == 200:

data = response.json()

print(data)

else:

print("Erro ao obter dados:", response.status\_code)

1. Método POST

import requests

url = "https://api.example.com/users"

data = {"name": "John Doe", "email": "johndoe@example.com"}

response = requests.post(url, json=data)

if response.status\_code == 201:

print("Usuário criado com sucesso.")

else:

print("Erro ao criar usuário:", response.status\_code)

1. Método PUT

import requests

url = "https://api.example.com/users/1"

data = {"name": "Jane Doe", "email": "janedoe@example.com"}

response = requests.put(url, json=data)

if response.status\_code == 200:

print("Usuário atualizado com sucesso.")

else:

print("Erro ao atualizar usuário:", response.status\_code)

1. Método DELETE

import requests

url = "https://api.example.com/users/1"

response = requests.delete(url)

if response.status\_code == 204:

print("Usuário excluído com sucesso.")

else:

print("Erro ao excluir usuário:", response.status\_code)

Estes são exemplos básicos de como usar os métodos GET, POST, PUT e DELETE com o pacote requests do Python. Eles mostram como fazer requisições HTTP para uma API REST e como lidar com as respostas dessas requisições.

## O que é extração de dados de uma API?

A extração de dados de uma API é o processo de coletar informações a partir de uma API. Isso é feito enviando uma requisição para a API, e a API responde com os dados solicitados.

Suponha que você tenha acesso a uma API incrementada do exemplo anterior que também permite consultar dados sobre a temperatura em uma determinada cidade. Para extrair esses dados, você precisa enviar uma requisição para a API, informando a cidade desejada. E a API, por sua vez, responderá com os dados de temperatura para a cidade solicitada.

# Autenticações em APIs

## Por que é importante compreender a autenticação em API's?

A autenticação é importante em API's porque ela garante que apenas usuários autorizados tenham acesso aos dados e funcionalidades fornecidos pela API. Além disso, a autenticação também ajuda a garantir a segurança dos dados, já que impede que pessoas não autorizadas possam acessá-los.

Por exemplo, imagine que você tenha acesso a uma API que permite coletar informações confidenciais, como dados financeiros. Sem autenticação, qualquer pessoa com conhecimento da API teria acesso a esses dados. Com autenticação, somente usuários autorizados poderão acessar e coletar essas informações.

Em Python, a autenticação em API's pode ser feita usando a biblioteca Requests e vários métodos, como Basic Authentication, OAuth, JWT, etc. É importante compreender a autenticação e saber como implementá-la corretamente para garantir a segurança dos dados coletados da API. Além disso, é importante saber como lidar com erros de autenticação, como por exemplo, quando as credenciais são inválidas ou expiradas.

Em caso de erro de autenticação, o ideal é verificar a documentação da API para entender o motivo do erro e como resolvê-lo. Algumas API's podem exigir que você renove as credenciais regularmente, enquanto outras podem ter limites de uso que precisam ser considerados. É importante também implementar códigos de tratamento de erros em sua aplicação para garantir que as informações sejam tratadas de maneira adequada em caso de erros de autenticação.

Em resumo, a compreensão da autenticação em API's é fundamental para garantir a segurança e a integridade dos dados coletados, além de permitir a integração de aplicativos e sistemas de maneira segura e controlada.

## O que é autenticação em API's?

Autenticação em API's é o processo de verificação de identidade de um usuário ou aplicativo antes de permitir o acesso aos dados ou recursos protegidos de uma API. É uma forma de garantir que somente usuários autorizados tenham acesso à informação e evitar acesso não autorizado ou malicioso.

## Exemplos básicos

Basic Authentication: É uma das formas mais simples de autenticação em API's. Nesta autenticação, o usuário envia seu nome de usuário e senha juntos com cada requisição na API. O servidor verifica se as credenciais são válidas antes de permitir o acesso aos dados ou recursos protegidos. Aqui está um exemplo em Python de como realizar a autenticação básica usando a biblioteca requests:  
import requests

url = "https://api.example.com/protected\_resource"

# Credenciais do usuário

username = "user"

password = "pass"

# Fazendo a requisição com autenticação básica

response = requests.get(url, auth=(username, password))

# Verificando o status da resposta

if response.status\_code == 200:

print("Acesso autorizado")

else:

print("Acesso não autorizado")

OAuth: É uma das formas mais utilizadas de autenticação em API's. Nesta autenticação, o usuário autoriza a aplicação a acessar seus dados protegidos através de uma API, sem compartilhar suas credenciais. Aqui está um exemplo em Python de como realizar a autenticação OAuth usando a biblioteca requests-oauthlib:  
import requests

from requests\_oauthlib import OAuth1

url = "https://api.example.com/protected\_resource"

# Chaves de acesso da aplicação

consumer\_key = "consumer\_key"

consumer\_secret = "consumer\_secret"

# Tokens de acesso do usuário

access\_token = "access\_token"

access\_token\_secret = "access\_token\_secret"

# Fazendo a requisição com autenticação OAuth

oauth = OAuth1(consumer\_key, consumer\_secret, access\_token, access\_token\_secret)

response = requests.get(url, auth=oauth)

# Verificando o status da resposta

if response.status\_code == 200:

print("Acesso autorizado")

else:

print("Acesso não autorizado")

JWT (JSON Web Tokens): É uma forma segura de transmitir informações através de requisições HTTP, como nome de usuário, ID de usuário, data de expiração, etc. O JWT é assinado digitalmente pelo servidor e pode ser verificado pelo cliente antes de permitir o acesso aos dados ou recursos protegidos. Aqui está um exemplo em Python de como realizar a autenticação JWT usando a biblioteca PyJWT:  
import jwt

import requests

url = "https://api.example.com/protected\_resource"

# Chaves de acesso da aplicação

secret\_key = "secret\_key"

# Dados do usuário a serem incluídos no JWT

payload = {"username": "user", "user\_id": 123}

# Codificando o JWT

token = jwt.encode(payload, secret\_key, algorithm='HS256')

# Adicionando o JWT à requisição

headers = {'Authorization': 'Bearer ' + token.decode('utf-8')}

response = requests.get(url, headers=headers)

# Verificando o status da resposta

if response.status\_code == 200:

print("Acesso autorizado")

else:

print("Acesso não autorizado")

## Funcionamento do processo de autenticação

1. Criação de chaves de acesso: As chaves de acesso são geradas pelo servidor da API e são fornecidas ao usuário ou aplicativo autorizado. Essas chaves são necessárias para autenticar as requisições à API.
2. Envio de credenciais na requisição: As credenciais do usuário, como nome de usuário, senha, tokens de acesso, etc., são enviadas com cada requisição à API. A forma de enviar as credenciais depende do tipo de autenticação escolhido.
3. Validação pelo servidor: O servidor da API valida as credenciais enviadas com a requisição antes de permitir o acesso aos dados ou recursos protegidos. Se as credenciais são válidas, o servidor retorna uma resposta de sucesso e os dados solicitados. Se as credenciais são inválidas, o servidor retorna uma resposta de erro e nega o acesso.

# Utilizando a biblioteca requests do Python

O pacote requests do Python é uma biblioteca que facilita a realização de requisições HTTP para API's. A instalação é feita através do comando pip install requests. Depois de instalada, ela pode ser usada para realizar requisições GET, POST, PUT, DELETE, entre outras, bastando importá-la com o comando import requests.

Um exemplo de requisição GET sem autenticação em Python é mostrado a seguir:

import requests

url = "https://api.example.com/dados"

response = requests.get(url)

print(response.status\_code)

print(response.json())

Neste exemplo, a requisição GET é feita para a URL https://api.example.com/dados e o retorno é armazenado na variável response. O código de status da resposta pode ser verificado com response.status\_code, e o corpo da resposta, que normalmente vem em formato JSON, pode ser acessado através de response.json().

Outro exemplo de requisição GET com autenticação básica, onde é necessário enviar usuario e senha na requisição, é mostrado a seguir:

import requests

url = "https://api.example.com/dados"

auth = requests.auth.HTTPBasicAuth("username", "password")

response = requests.get(url, auth=auth)

print(response.status\_code)

print(response.json())

Neste exemplo, a autenticação é feita através da classe HTTPBasicAuth, que recebe como parâmetros o usuário e a senha. O objeto auth é passado como parâmetro na requisição GET com o nome auth.

Um exemplo requisição GET com OAuth, usando a biblioteca OAuthLib, é mostrado a seguir:

import requests

from requests\_oauthlib import OAuth1

url = "https://api.example.com/dados"

auth = OAuth1("consumer\_key", "consumer\_secret", "access\_token", "access\_token\_secret")

response = requests.get(url, auth=auth)

print(response.status\_code)

print(response.json())

Neste exemplo, a autenticação é feita através da classe OAuth1 da biblioteca requests\_oauthlib, que recebe como parâmetros o "consumer key", o "consumer secret", o "access token" e o "access token secret". O objeto auth é passado como parâmetro na requisição GET com o nome auth.

# Tratamento de Erros na Extração de Dados de API's

É importante tratar possíveis erros que possam ocorrer durante a extração de dados de uma API. Algumas dicas para tratar erros incluem:

1. Verificação de Conexão: verifique se você está conectado à internet antes de fazer uma requisição;
2. Verificação de URL: verifique se a URL da API está correta e se o endpoint existe antes de fazer a requisição; e

Verificação de Status da Resposta: o status da resposta pode ser utilizado para verificar se a requisição foi bem-sucedida ou não. Em Python, é possível verificar o status da resposta com o método response.status\_code. Por exemplo:  
import requests

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

if response.status\_code == 200:

print("A requisição foi bem-sucedida.")

else:

print("A requisição falhou. Código de status: ", response.status\_code)

Tratamento de Exceções: é possível utilizar o try e except para tratar possíveis erros durante a requisição. Por exemplo:  
import requests

url = "https://api.example.com/data"

try:

response = requests.get(url)

response.raise\_for\_status()

except requests.exceptions.HTTPError as err:

print("Ocorreu um erro na requisição: ", err)

1. Neste exemplo, fazemos a requisição GET ao endereço url, e usamos o método raise\_for\_status para verificar se a resposta tem algum erro HTTP. Em caso positivo, é gerado um erro HTTPError que é capturado pelo bloco except e impressa uma mensagem de erro. Caso haja algum outro erro na requisição, é gerada uma exceção RequestException que também é capturada e imprime uma mensagem de erro.

Além disso, é importante sempre checar as documentações das API's a serem usadas para conhecer os possíveis erros e códigos de status que podem ser retornados e tratá-los adequadamente no seu código.

Tratamento de Resposta Vazia: verifique se a resposta da API está vazia antes de manipulá-la. Se a resposta estiver vazia, é possível exibir uma mensagem de erro ou tratar o erro de outra forma. Por exemplo:  
import requests

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

if not response.json():

print("A resposta da API está vazia.")

else:

# manipulação dos dados aqui

print(response.json())

Além disso, é importante sempre checar as documentações das API's a serem usadas para conhecer os possíveis erros e códigos de status que podem ser retornados e tratá-los adequadamente no seu código.

# Tratamento dos dados retornados por API’s

O tratamento de dados retornados pela API é uma etapa importante na extração de dados. O objetivo desse tratamento é transformar a informação retornada pela API em um formato que seja mais fácil de manipular para o desenvolvedor. O formato mais comum de retorno é o JSON (JavaScript Object Notation), que é um formato de dados leve e fácil de ler.

Para tratar os dados retornados em Python, é possível utilizar a biblioteca json. Essa biblioteca possui a função loads, que permite transformar o conteúdo de uma string em um objeto Python (geralmente um dicionário).

import json

# Exemplo de resposta da API em formato JSON

api\_response = '{"name": "John", "age": 32, "city": "New York"}'

# Transformação da string em objeto Python

parsed\_response = json.loads(api\_response)

# Acesso aos dados do objeto

print("Name:", parsed\_response["name"])

print("Age:", parsed\_response["age"])

print("City:", parsed\_response["city"])

Além da biblioteca json, é possível utilizar bibliotecas mais avançadas como o pandas para manipulação de dados. Este pacote é uma biblioteca poderosa que permite trabalhar com dados em formato tabular, como planilhas do Excel. É possível transformar um objeto retornado pela API em um DataFrame do pandas, o que permite realizar operações avançadas de manipulação de dados, como filtros, agrupamentos e cálculos.

import pandas as pd

import json

# Exemplo de resposta da API em formato JSON

api\_response = '[{"name": "John", "age": 32, "city": "New York"}, {"name": "Jane", "age": 28, "city": "London"}]'

# Transformação da string em objeto Python

parsed\_response = json.loads(api\_response)

# Transformação do objeto em DataFrame do pandas

df = pd.DataFrame(parsed\_response)

# Exibição do DataFrame

print(df)

Em resumo, o tratamento de dados retornados pela API permite transformar a informação retornada pela API em um formato mais fácil de manipular, como objetos Python ou DataFrames do pandas. Isso permite realizar operações mais avançadas de manipulação de dados, tornando a extração de dados de uma API muito mais simples e eficiente.

## Tópicos extras sobre API’s

A manipulação de consulta em APIs pode ser feita com o uso de diferentes técnicas, incluindo filtro, ordenação e paginação. A seguir, iremos detalhar algumas destas principais técnicas:

Filtragem: é a técnica usada para selecionar apenas os dados relevantes para a sua consulta. Isso pode ser feito por meio de parâmetros de consulta específicos na URL da API. Por exemplo, se você deseja recuperar todos os registros que contêm uma determinada palavra-chave, você pode adicionar esse termo à URL da API como um parâmetro de consulta.  
import requests

url = "https://api.example.com/records?keyword=python"

response = requests.get(url)

data = response.json()

Ordenação: é a técnica usada para classificar os dados de acordo com algum critério específico. Isso também pode ser feito por meio de parâmetros de consulta na URL da API. Por exemplo, se você deseja classificar os registros por data, você pode adicionar o parâmetro "order\_by" à URL da API com o valor "data".  
import requests

url = "https://api.example.com/records?order\_by=date"

response = requests.get(url)

data = response.json()

Paginação: é a técnica usada para dividir grandes quantidades de dados em pequenas páginas para melhorar a performance e a usabilidade. Isso também pode ser feito por meio de parâmetros de consulta na URL da API. Por exemplo, se você deseja recuperar apenas a primeira página de registros, você pode adicionar os parâmetros "page" e "page\_size" à URL da API.  
import requests

url = "https://api.example.com/records?page=1&page\_size=10"

response = requests.get(url)

data = response.json()

Em resumo, a manipulação de consulta em APIs incluindo filtro, ordenação e paginação em Python pode ser realizada usando parâmetros de consulta na URL da API e na biblioteca "requests" para fazer a solicitação à API e processar a resposta.

### Error code 429: Too many requests

Ao fazer consultas frequentes a uma API, é comum que você receba um erro "429 Too Many Requests" indicando que você atingiu o limite de solicitações por período de tempo. Para lidar com esse erro, você pode implementar uma rotina de retentativas que aguarde um período de tempo antes de tentar novamente a consulta. Aqui está um exemplo de como isso pode ser feito em Python:

import time

import requests

# Função decoradora para realizar retentativas em caso de erro 429

def retry\_on\_429(func):

# Número máximo de retentativas

MAX\_ATTEMPTS = 5

# Loop para retentativas

for attempt in range(1, MAX\_ATTEMPTS + 1):

# Realiza a requisição

response = func()

# Verifica se a resposta não é 429

if response.status\_code != 429:

# Retorna a resposta caso seja diferente de 429

return response

# Tempo de espera indicado no cabeçalho da resposta "Retry-After"

wait\_time = response.headers.get("Retry-After", 1)

print(f"Too many requests, waiting {wait\_time} seconds before retrying")

# Espera o tempo indicado antes de retentar

time.sleep(wait\_time)

# Lança exceção caso o número máximo de retentativas seja excedido

raise Exception("Exceeded maximum number of retries")

# Função para realizar a requisição à API

@retry\_on\_429

def make\_request():

# URL da API

url = "https://api.example.com/data"

# Cabeçalhos da requisição

headers = {

"Authorization": "Bearer xxx",

}

# Realiza a requisição

response = requests.get(url, headers=headers)

# Retorna a resposta

return response

# Realiza a requisição com retentativas

response = make\_request()

# Processa a resposta

data = response.json()

Neste exemplo, a função make\_request é responsável por fazer a solicitação à API. A função retry\_on\_429 é uma função decoradora que irá retentar a chamada à função make\_request até MAX\_ATTEMPTS vezes se o status da resposta for 429. Se o status da resposta for diferente de 429, a resposta será retornada. Caso o número máximo de retentativas seja excedido, uma exceção será lançada.

A rotina de retentativas usa o tempo de espera indicado no cabeçalho da resposta "Retry-After", que informa o tempo em segundos que você deve esperar antes de tentar novamente a solicitação. Se esse cabeçalho não estiver presente, a rotina assume um tempo de espera padrão de 1 segundo.

## Conclusão

Neste tutorial, aprendemos sobre o que é uma API, a extração de dados de uma API, a importância da compreensão da autenticação em API's, os tipos de autenticação e como eles funcionam em Python, usando bibliotecas como requests e OAuthLib. Além disso, aprendemos como tratar os dados retornados pela API, usando bibliotecas com Pandas, e como tratar erros e exceções na requisição à API.

É importante destacar que as API's são uma das principais formas de se acessar dados em larga escala e automatizar tarefas, e compreender como trabalhar com elas é essencial para muitos profissionais de tecnologia.

Espero que este tutorial tenha sido útil e ajude a compreender melhor o funcionamento de API's e como trabalhar com elas em Python.

# Referências

As seguintes referências podem ser úteis para aprender mais sobre o assunto de manipulação de consultas API com Python e fornecerão informações detalhadas sobre como usar o módulo requests em Python e incluem exemplos de código e explicações detalhadas.

* Guia para iniciantes em APIs: <https://restfulapi.net/>
* Documentação da biblioteca Requests (<http://docs.python-requests.org/en/master/>): contém informações detalhadas sobre como usar a biblioteca para realizar requisições HTTP.
* Tutorial básico sobre o módulo requests: <https://realpython.com/python-requests/>
* Guia de boas práticas para uso de APIs (<https://docs.microsoft.com/en-us/azure/architecture/best-practices/api-design>): inclui recomendações sobre como lidar com erros e limitações de rate limit em APIs.
* Artigo sobre como fazer requisições HTTP com o módulo requests: <https://towardsdatascience.com/parallel-web-requests-in-python-4d30cc7b8989>
* Tutorial sobre funções decoradoras em Python (<https://realpython.com/primer-on-python-decorators/>): explica o conceito de funções decoradoras e como elas podem ser usadas para modificar o comportamento de outras funções.
* Curso digital Ada: [Introdução a Python](https://cursos.letscode.com.br/curso-digital/ad77ffa3-6dde-4efb-9e0d-3b14e60b097b)
* Curso digital Ada: [API REST](https://cursos.letscode.com.br/curso-digital/4c455c4a-51e8-4ae1-a3ec-b2b2e26bcf87)