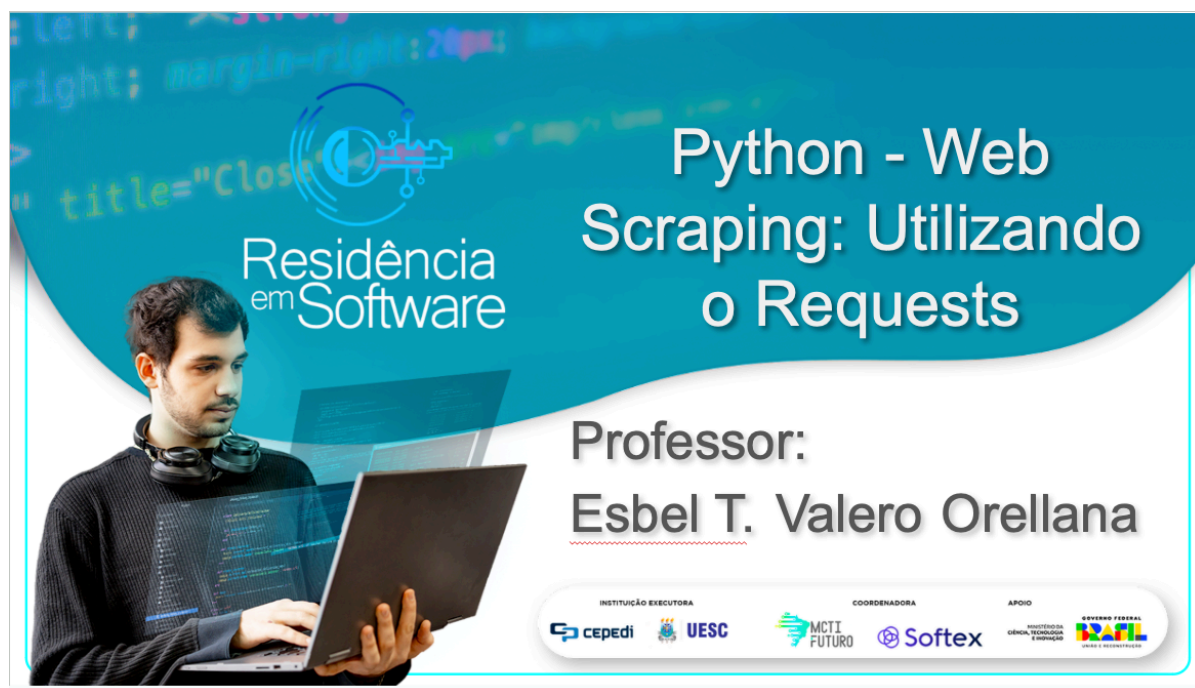


Módulo de Programação Python

Trilha Python - Aula 37/38: Web Scraping: Utilizando o Requests



Residência em Software

Python - Web Scraping: Utilizando o Requests

Professor:
Esbel T. Valero Orellana

INSTITUIÇÃO EXECUTORA
cepedi UESC

COORDENADORA
MCTI FUTURO Softex

APOIO
GOVERNO FEDERAL
Ministério da Ciência, Tecnologia e Inovação

Pegando os dados direto da Internet

```

In [1]: 1 import pandas as pd
        2 url = "https://www.stats.govt.nz/assets/Uploads/Business-employ
        3 url = "https://portal.inmet.gov.br/uploads/dadoshistoricos/2001
        4 train = pd.read_table(url)
        5 train.head()

624 def _safe_read(self, amt):
625     """Read the number of bytes requested.
626
627     This function should be used when <amt> bytes "shoul
d" be present for
628     reading. If the bytes are truly not available (due t
o EOF), then the
629     IncompleteRead exception can be used to detect the p
roblem.
630     """
--> 631     data = self.fp.read(amt)
632     if len(data) < amt:
633         raise IncompleteRead(data, amt-len(data))

File /opt/anaconda3/envs/httpPy/lib/python3.10/socket.py:705, in
SocketIO.readinto(self, b)
703 while True:
704     try:
--> 705         return self._sock.recv_into(b)
706     except timeout:

```

Uma introdução à biblioteca Requests de Python

A biblioteca **Requests** foi desenvolvida com o objetivo de simplificar a realização de solicitações **HTTP** para servidores *web* e o trabalho tratar as respostas a estas solicitações.

A biblioteca de **Requests** é o padrão de fato para fazer solicitações **HTTP** em **Python**. Ela abstrai as complexidades de fazer solicitações por trás de uma **API** simples e bonita para que você possa se concentrar na interação com serviços e no consumo de dados em seu aplicativo.

Os métodos implementados na biblioteca **Requests** executam operações **HTTP** em um servidor *web* especificado pela sua **URL**. Eles também suportam o envio de informações extras para o servidor por meio de parâmetros e cabeçalhos, codificando as respostas do servidor, detectando erros e manipulando redirecionamentos.

Além de simplificar a forma como trabalhamos com as operações **HTTP**, a biblioteca **Requests** fornece alguns recursos avançados, como tratamento de exceções **HTTP** e autenticação.

```
In [2]: 1 #pip install requests
```

Depois que o **Requests** for instalado basta importar o pacote para usar seus recursos.

```
In [3]: 1 import requests
```

Usando GET

Métodos **HTTP**, como **GET** e **POST**, determinam qual ação você está tentando executar ao fazer uma solicitação **HTTP**.

Além destas solicitações, existem vários outros métodos comuns que usaremos posteriormente.

Um dos métodos **HTTP** mais comuns é **GET**. O método **GET** indica que você está tentando obter ou recuperar dados de um determinado recurso. Para fazer uma solicitação **GET**, se utiliza `requests.get()`.

Vamos testar fazendo uma solicitação **GET** para a **API Root REST** do **GitHub** chamando `get()`.

```
In [ ]: 1 requests.get('https://api.github.com')
```

A resposta

Uma resposta é um objeto poderoso que permite inspecionar os resultados da solicitação.

Podemos fazer a mesma solicitação novamente, mas desta vez armazenamos o valor de retorno em uma variável para que você possa ver com mais detalhes seus atributos e comportamentos.

```
In [ ]: 1 resposta = requests.get('https://api.github.com')
```

```
In [ ]: 1 type(resposta)
```

Códigos de status

A primeira informação que você pode coletar do `Response` é o código de status. Um código de status informa sobre o status da solicitação.

Por exemplo, um status 200 OK significa que sua solicitação foi bem-sucedida, enquanto um status 404 NOT FOUND significa que o recurso que você estava procurando não foi encontrado.

Existem muitos outros códigos de status possíveis para fornecer informações específicas sobre o que aconteceu com sua solicitação.

Acessando `.status_code`, você pode ver o código de status que o servidor retornou.

```
In [ ]: 1 resposta.status_code
```

```
In [ ]: 1 data2001 = requests.get(url)
```

```
In [ ]: 1 requests.get("https://portal.inmet.gov.br/dadoshistoricos")
```

O `.status_code` retornou 200, o que significa que sua solicitação foi bem-sucedida e o servidor respondeu com os dados que você estava solicitando.

Às vezes, você pode querer usar essas informações para tomar decisões em seu código.

```
In [ ]: 1 if resposta.status_code == 200:
2     print('Solicitação atendida!')
3 elif resposta.status_code == 404:
4     print('Não encontrado.')
```

Entretanto, se você usar uma instância `Response` em uma expressão condicional, ela será avaliada como `True` se o código de status estiver entre 200 e 400 e como `False` caso contrário.

```
In [ ]: 1 if resposta:
2     print('Solicitação atendida!')
3 else:
4     print('Não encontrado.')
```

Lembre-se de que este método não verifica se o código de status é igual a 200. A razão para isso é que outros códigos de status na faixa de 200 a 400, como 204 NO CONTENT e 304 NOT MODIFIED, também são considerados bem-sucedidos no sentido que eles fornecem alguma resposta viável.

Por exemplo, o 204 informa que a resposta foi bem-sucedida, mas não há conteúdo para retornar no corpo da mensagem.

Por outro lado, se você não quer verificar o código de status da resposta em uma instrução if. Em vez disso, você deseja gerar uma exceção se a solicitação não tiver êxito. Você pode fazer isso usando método `.raise_for_status()` :

```
In [ ]: 1 from requests.exceptions import HTTPError
        2
        3 for url in ['https://api.github.com', 'https://api.github.com/i
        4     try:
        5         resposta = requests.get(url)
        6         resposta.raise_for_status()
        7     except HTTPError as http_err:
        8         print(f'Aconteceu um erro HTTP: {http_err}')
        9     except Exception as err:
       10         print(f'Aconteceu outro tipo de erro: {err}')
       11     else:
       12         print('Solicitação atendida!')
```

Se você utiliza o método `.raise_for_status()` , um `HTTPError` será gerado para determinados códigos de status. Se o código de status indicar uma solicitação bem-sucedida, o programa continuará sem que essa exceção seja gerada.

Já temos um boa ideia de como lidar com o código de status da resposta recebida do servidor. Entretanto, a parte mais importante da resposta do servidor costuma ser o conteúdo da mesma.

Conteúdo

A resposta de uma solicitação `GET` geralmente contém algumas informações valiosas, conhecidas como carga útil, no corpo da mensagem. Usando os atributos e métodos de `Response` , você pode visualizar a carga em vários formatos diferentes.

```
In [ ]: 1 resposta = requests.get('https://api.github.com')
        2 resposta.content
```

```
In [ ]: 1 type(resposta.content)
```

Embora o atributo `.content` forneça acesso aos bytes brutos da carga útil da resposta, muitas vezes você desejará convertê-los em uma string usando uma codificação de caracteres como UTF-8. A classe `Response` fornece este tipo de informação no atributo `.text`.

```
In [ ]: 1 resposta.text
```

```
In [ ]: 1 type(resposta.text)
```

Como a decodificação de bytes para um `str` requer um esquema de codificação, as requests tentarão adivinhar a codificação com base nos cabeçalhos da resposta, se você não especificar um. Você pode fornecer uma codificação explícita definindo `.encoding` antes de acessar `.text`.

```
In [ ]: 1 resposta.encoding # o encoding definido no cabeçalho da respos
```

```
In [ ]: 1 # Neste outro exeplo
2 url = "https://portal.inmet.gov.br/uploads/dadoshistoricos/"
3 dataTem = requests.get(url)
4 dataTem.encoding
```

```
In [ ]: 1 dataTem.text
```

```
In [ ]: 1 # Neste caso podemos mudar a codificação
2 dataTem.encoding = 'UTF-8'
```

```
In [ ]: 1 dataTem.text
```

Se você reparar na resposta, verá que na verdade o conteúdo é um **JSON** serializado. Para obter um dicionário, você pode pegar o `str` recuperado de `.text` e transformá-lo usando `json.loads()`.

Porém, uma maneira mais simples de realizar esta tarefa é usar método `.json()`.

```
In [ ]: 1 import json
2 meuJson = json.loads(resposta.text)
3 meuJson
```

```
In [ ]: 1 resposta.json()
```

O tipo de valor de retorno de `.json()` é um dicionário, então você pode acessar valores no objeto por chave.

Se você precisar de mais informações, como metadados sobre a resposta em si, você precisará examinar os cabeçalhos da resposta.

Cabeçalhos

Os cabeçalhos de resposta podem fornecer informações úteis, como o tipo de conteúdo da carga útil da resposta e um limite de tempo para armazenar a resposta em cache. Para visualizar esses cabeçalhos, acesse `.headers`.

```
In [ ]: 1 resposta.headers
```

Repare que o `.headers` retorna um objeto semelhante a um dicionário, permitindo acessar valores de cabeçalho por chave. Por exemplo, para ver o tipo de conteúdo da carga útil da resposta, você pode acessar `Content-Type`.

```
In [ ]: 1 resposta.headers['content-type']
```

Diferente de um dicionário, se você usa a chave `'content-type'` ou `'Content-Type'`, obterá o mesmo valor.

Parâmetros de string de consulta

Uma maneira comum de personalizar uma solicitação `GET` é passar valores por meio de parâmetros de *string* de consulta na **URL**. Para fazer isso usando `get()`, você passa dados para parâmetros. Por exemplo, você pode usar a **API** de pesquisa do **GitHub** para procurar o repositório do curso.

```
In [ ]: 1 # Search GitHub's repositories for ResTIC18
2 resposta = requests.get(
3     'https://api.github.com/search/repositories',
4     params={'q': 'ResTIC18'},
5 )
6
7 # Inspect some attributes of the `ResTIC18` repository
8 json_response = resposta.json()
9 repository = json_response['items'][0]
10 print(f'Repository name: {repository["name"]}') # Python 3.6+
11 print(f'Repository description: {repository["description"]}')
```

Ao passar o dicionário {'q': 'ResTIC18'} para o parâmetro `params` do método `.get()`, você pode modificar os resultados que retornam da **API** de pesquisa.

Você pode passar parâmetros para `get()` na forma de um dicionário, como acabou de fazer, ou como uma lista de tuplas.

```
In [ ]: 1 requests.get('https://api.github.com/search/repositories', param
```

Ou ainda, pode até passar os valores como bytes.

```
In [ ]: 1 requests.get('https://api.github.com/search/repositories', para
```

Strings de consulta são úteis para parametrizar requests `GET`. Você também pode personalizar suas solicitações adicionando ou modificando os cabeçalhos enviados.

Cabeçalhos do request

Para personalizar cabeçalhos, você passa um dicionário de cabeçalhos **HTTP** para `get()` usando o parâmetro `headers`. Por exemplo, você pode alterar sua solicitação de pesquisa anterior para destacar termos de pesquisa correspondentes nos resultados, especificando o tipo de mídia de correspondência de texto no cabeçalho `Accept`.

```
In [ ]: 1 resposta = requests.get('https://api.github.com/search/reposito
2                                params={'q': 'ResTIC18'},
3                                headers={'Accept': 'application/vnd.git
4                                )
5
6 json_response = resposta.json()
7 repository = json_response['items'][0]
8 print(f'Text matches: {repository["text_matches"]}')

```

O cabeçalho `Accept` informa ao servidor quais tipos de conteúdo seu aplicativo pode manipular. Nesse caso, como você espera que os termos de pesquisa correspondentes sejam destacados, você está usando o valor do cabeçalho `application/vnd.github.v3.text-match+json`, que é um cabeçalho proprietário do **GitHub** `Accept` onde o conteúdo é um formato **JSON** especial.

Outros métodos HTTP

Além de `GET`, outros métodos **HTTP** populares incluem `POST`, `PUT`, `DELETE`, `HEAD`, `PATCH` e `OPTIONS`. A **Requests** fornece um método, com uma assinatura semelhante a `get()`, para cada um destes métodos **HTTP**.


```
In [ ]: 1 requests.post('https://httpbin.org/post', data={'key': 'value'})
        2 requests.put('https://httpbin.org/put', data={'key': 'value'})
        3 requests.delete('https://httpbin.org/delete')
        4 requests.head('https://httpbin.org/get')
        5 requests.patch('https://httpbin.org/patch', data={'key': 'value'})
        6 requests.options('https://httpbin.org/get')
```

Cada chamada de função faz uma solicitação ao serviço [httpbin \(https://httpbin.org/\)](https://httpbin.org/) usando o método **HTTP** correspondente. Para cada método, você pode inspecionar as respostas da mesma forma que fez antes.

```
In [ ]: 1 resposta = requests.post('https://httpbin.org/post', data={'key': 'value'})
        2 resposta.json()
```

Cabeçalhos, corpos de resposta, códigos de status e muito mais são retornados na Resposta para cada método. Vamos entender melhor como funcionam os métodos POST , PUT e PATCH .

O corpo da mensagem

De acordo com a especificação **HTTP**, as solicitações POST , PUT e PATCH menos comuns passam seus dados por meio do corpo da mensagem em vez de por meio de parâmetros na string de consulta. Usando **Requests**, você passará a carga para o parâmetro `data` da função correspondente.

O parâmetro `data` leva um dicionário, uma lista de tuplas, bytes ou um objeto semelhante a um arquivo. Você desejará adaptar os dados enviados no corpo da sua solicitação às necessidades específicas do serviço com o qual está interagindo.

Por exemplo, se o tipo de conteúdo da sua solicitação for `application/x-www-form-urlencoded` , você poderá enviar os dados do formulário como um dicionário.

```
In [ ]: 1 requests.post('https://httpbin.org/post', data={'key': 'value'})
```

Você também pode enviar os mesmos dados como uma lista de tuplas.

```
In [ ]: 1 requests.post('https://httpbin.org/post', data=[('key', 'value')])
```

Se, no entanto, você precisar enviar dados **JSON**, poderá usar o parâmetro `json`. Quando você passa dados via `json`, as solicitações serializarão seus dados e adicionarão o cabeçalho `Content-Type` correto para você.

O site httpbin.org é um ótimo recurso criado pelo autor de **Requests**, Kenneth Reitz. É um serviço que aceita solicitações de teste e responde com dados sobre as solicitações. Por exemplo, você pode usá-lo para inspecionar uma solicitação POST básica.

```
In [ ]: 1 resposta = requests.post('https://httpbin.org/post', json={'key': 'value'})
        2 json_response = resposta.json()
        3 json_response['data']
```

```
In [ ]: 1 json_response['headers']['Content-Type']
```

Inspecionando seu request

Quando você faz uma solicitação, a biblioteca de **Requests** prepara a solicitação antes de enviá-la ao servidor de destino. A preparação da solicitação inclui itens como validação de cabeçalhos e serialização de conteúdo **JSON**.

Você pode visualizar o `PreparedRequest` acessando `.request`.

```
In [ ]: 1 response = requests.post('https://httpbin.org/post', json={'key': 'value'})
        2 response.request.headers['Content-Type']
```

```
In [ ]: 1 response.request.url
```

```
In [ ]: 1 response.request.body
```

A inspeção do `PreparedRequest` dá acesso a todos os tipos de informações sobre a solicitação que está sendo feita, como carga útil, URL, cabeçalhos, autenticação e muito mais.

Muitos serviços que você encontra no dia a dia, exigirão que você se autentique de alguma forma.

Autenticação

A autenticação ajuda um serviço a entender quem você é. Normalmente, você fornece suas credenciais a um servidor passando dados por meio do cabeçalho

`Authorization` ou de um cabeçalho personalizado definido pelo serviço. Todas as funções de solicitação que você viu até agora fornecem um parâmetro chamado `auth`, que permite passar suas credenciais.

Um exemplo de **API** que requer autenticação é a **API** de usuário autenticado do GitHub. Este endpoint fornece informações sobre o perfil do usuário autenticado. Para fazer uma solicitação à API de usuário autenticado, você pode passar seu nome de usuário e senha do GitHub em uma tupla para `get()`.

```
In [ ]: 1 from getpass import getpass
        2 requests.get('https://api.github.com/user', auth=('etvorellana'
        3
```

A solicitação será bem-sucedida se as credenciais que você passou na tupla para autenticação forem válidas. Se você tentar fazer esta solicitação sem credenciais, verá que o código de status é 401 Não autorizado.

Quando você passa seu nome de usuário e senha em uma tupla para o parâmetro `auth`, as solicitações aplicam as credenciais usando o esquema de autenticação de acesso básico do HTTP nos bastidores.

Portanto, você poderia fazer a mesma solicitação passando credenciais de autenticação Básica explícitas usando `HTTPBasicAuth`.

```
In [ ]: 1 from requests.auth import HTTPBasicAuth
        2 from getpass import getpass
        3 requests.get('https://api.github.com/user', auth=HTTPBasicAuth(
```

Embora não seja necessário ser explícito para a autenticação Básica, você pode querer autenticar usando outro método. A **Requests** fornece outros métodos de autenticação prontos para uso, como `HTTPDigestAuth` e `HTTPProxyAuth`.

Você pode até fornecer seu próprio mecanismo de autenticação. Para fazer isso, primeiro você deve criar uma subclasse de `AuthBase`. Então, você implementa `__call__()` dessa classe.

```
In [ ]: 1 from requests.auth import AuthBase
        2
        3 class TokenAuth(AuthBase):
        4     """Implements a custom authentication scheme."""
        5
        6     def __init__(self, token):
        7         self.token = token
        8
        9     def __call__(self, r):
        10         """Attach an API token to a custom auth header."""
        11         r.headers['X-TokenAuth'] = f'{self.token}'
        12         return r
        13
        14
        15 requests.get('https://httpbin.org/get', auth=TokenAuth('12345ab
```

Aqui, seu mecanismo `TokenAuth` personalizado recebe um token e, em seguida, inclui esse token no cabeçalho `X-TokenAuth` da sua solicitação.

Mecanismos de autenticação incorretos podem levar a vulnerabilidades de segurança; portanto, a menos que um serviço exija um mecanismo de autenticação personalizado por algum motivo, você sempre desejará usar um esquema de autenticação testado e comprovado, como `Basic`.

Verificação de certificado SSL

Sempre que os dados que você está tentando enviar ou receber forem confidenciais, a segurança é importante. A maneira como você se comunica com sites seguros por HTTP é estabelecendo uma conexão criptografada usando SSL, o que significa que a verificação do certificado SSL do servidor de destino é fundamental.

A boa notícia é que `requests` faz isso por padrão. No entanto, há alguns casos em que você pode querer alterar esse comportamento.

Se você deseja desabilitar a verificação do Certificado SSL, você passa `False` para o parâmetro `verify` da função `request`.

```
In [4]: 1 requests.get('https://api.github.com', verify=False)
```

```
/opt/anaconda3/envs/httpPy/lib/python3.10/site-packages/urllib3/co
nnectionpool.py:1103: InsecureRequestWarning: Unverified HTTPS req
uest is being made to host 'api.github.com'. Adding certificate ve
rification is strongly advised. See: https://urllib3.readthedocs.io/en/latest/advanced-usage.html#tls-warnings (https://urllib3.readthedocs.io/en/latest/advanced-usage.html#tls-warnings)
  warnings.warn(
```

```
Out[4]: <Response [200]>
```

Desempenho

Ao usar solicitações, especialmente em um ambiente de aplicação de produção, é importante considerar as implicações de desempenho. Recursos como controle de tempo limite, sessões e limites de novas tentativas podem ajudar você a manter seu aplicativo funcionando perfeitamente.

Tempos limite

Quando você faz uma solicitação inline a um serviço externo, seu sistema precisará aguardar a resposta antes de prosseguir. Se o seu aplicativo esperar muito por essa resposta, as solicitações ao seu serviço poderão sofrer backup, a experiência do usuário poderá ser prejudicada ou os trabalhos em segundo plano poderão travar.

Por padrão, as solicitações aguardarão indefinidamente pela resposta, portanto, você quase sempre deve especificar um tempo limite para evitar que essas coisas aconteçam. Para definir o tempo limite da solicitação, use o parâmetro `timeout`. o tempo limite pode ser um número inteiro ou flutuante que representa o número de segundos de espera por uma resposta antes do tempo limite: