Requests в Python – Примеры выполнения HTTP запросов

Ключевые аспекты инструкции:

- **Создание запросов** при помощи самых популярных HTTP методов;
- **Редактирование** заголовков запросов и данных при помощи строки запроса и содержимого сообщения;
- **Анализ** данных запросов и откликов;
- Создание **авторизированных** запросов;
- **Настройка** запросов для предотвращения сбоев и замедления работы приложения.

Python библиотека Requests метод GET

Такие HTTP методы, как `GET` и `POST`, определяют, какие действия будут выполнены при создании **HTTP запроса**. Помимо `GET` и `POST` для этой задачи могут быть использованы некоторые другие методы. Далее они также будут описаны в руководстве.
`GET` является одним из самых популярных HTTP методов. Метод `GET` указывает на то, что происходит попытка извлечь данные из определенного ресурса. Для того, чтобы выполнить запрос `GET`, используется `requests.get()`.

Для проверки работы команды будет выполнен запрос `GET` в отношении Root REST API на GitHub. Для указанного ниже URL вызывается метод `get()`.

```
requests.get('https://api.github.com')
<Response [200]>
```

Объект Response получение ответа на запрос в Python

`Response` представляет собой довольно мощный объект для анализа результатов запроса. В качестве примера будет использован предыдущий запрос, только на этот раз результат будет представлен в виде переменной. Таким образом, получится лучше изучить его атрибуты и особенности использования.

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

В данном примере при помощи `get()` захватывается определенное значение, что является частью объекта `Response`, и помещается в переменную под названием `response`. Теперь можно использовать переменную `response` для того, чтобы изучить данные, которые были получены в результате запроса `GET`.

Попробуем получить веб-страницу. В этом примере давайте рассмотрим общий тайм-лайн GitHub:

```
r = requests.get('https://api.github.com/events')
```

Мы получили объект Response с именем r. С помощью этого объекта можно получить всю необходимую информацию.

Простой API Requests означает, что все формы HTTP запросов- очевидны. Ниже приведен пример того, как вы можете сделать запрос HTTP POST:

```
r = requests.post('http://httpbin.org/post', data = {'key':'value'})
```

HTTP коды состояний

Самыми первыми данными, которые будут получены через `Response`, будут коды состояния. **Коды состояния** сообщают о статусе запроса.
Например, статус `200 OK` значит, что запрос успешно выполнен. А вот статус `404 NOT FOUND` говорит о том, что запрашиваемый ресурс не был найден. Существует множество других статусных кодов, которые могут сообщить важную информацию, связанную с запросом.
Используя `.status_code`, можно увидеть код состояния, который возвращается с сервера.

```
>>> response.status_code
200
```

`.status_code` вернул значение `200`. Это значит, что запрос был выполнен успешно, а сервер ответил, отобразив запрашиваемую информацию.
В некоторых случаях необходимо использовать полученную информацию для написания программного кода.

```
if response.status_code == 200:
    print('Success!')
elif response.status_code == 404:
    print('Not Found.')
```

Если использовать Response в условных конструкциях, то при получении кода состояния в промежутке от 200 до 400, будет выведено значение True. В противном случае отобразится значение False.

Последний пример можно упростить при помощи использования оператора if.

```
if response:
    print('Success!')
else:
    print('An error has occurred.')
```

Стоит иметь в виду, что данный способ не проверяет, имеет ли статусный код точное значение 200. Причина заключается в том, что другие коды в промежутке от 200 до 400, например, 204 NO CONTENT и 304 NOT MODIFIED, также считаются успешными в случае, если они могут предоставить действительный ответ.

К примеру, код состояния 204 говорит о том, что ответ успешно получен, однако в полученном объекте нет содержимого. Можно сказать, что для оптимально эффективного использования способа необходимо убедиться, что начальный запрос был успешно выполнен. Требуется изучить код состояния и в случае необходимости произвести необходимые поправки, которые будут зависеть от значения полученного кода.

Допустим, если при использовании оператора if вы не хотите проверять код состояния, можно расширить диапазон исключений для неудачных результатов запроса. Это можно сделать при помощи использования .raise_for_status().

```
import requests
from requests.exceptions import HTTPError

for url in ['https://api.github.com', 'https://api.github.com/invalid']:
    try:
        response = requests.get(url)

        # если ответ успешен, исключения задействованы не будут
        response.raise_for_status()
    except HTTPError as http_err:
        print(f'HTTP error occurred: {http_err}')  # Python 3.6
    except Exception as err:
        print(f'Other error occurred: {err}')  # Python 3.6
    else:
        print('Success!')
```

В случае вызова исключений через .raise_for_status() к некоторым кодам состояния применяется HTTPError. Когда код состояния показывает, что запрос успешно выполнен, программа продолжает работу без применения политики исключений.

Получить содержимое страницы в Requests

Зачастую ответ на запрос GET содержит весьма ценную информацию. Она находится в теле сообщения и называется пейлоад (payload). Используя атрибуты и методы библиотеки Response, можно получить пейлоад в различных форматах.

Для того, чтобы получить содержимое запроса в байтах, необходимо использовать .content.

```
>>> response = requests.get('https://api.github.com')
>>> response.content
```

Использование .content обеспечивает доступ к чистым байтам ответного пейлоада, то есть к любым данным в теле запроса. Однако, зачастую требуется конвертировать полученную информацию в строку в кодировке UTF-8. response делает это при помощи .text.

```
>>> response.text
```

Декодирование байтов в строку требует наличия определенной модели кодировки. По умолчанию requests попытается узнать текущую кодировку, ориентируясь по заголовкам HTTP. Указать необходимую кодировку можно при помощи добавления .encoding перед .text.

```
>>> response.encoding = 'utf-8' # Optional: requests
infers this internally
>>> response.text
```

HTTP заголовки в Requests

HTTP заголовки ответов на запрос могут предоставить определенную полезную информацию. Это может быть тип содержимого ответного пейлоада, а также ограничение по времени для кеширования ответа. Для просмотра HTTP заголовков загляните в атрибут .headers.

```
>>> response.headers
```

.headers возвращает словарь, что позволяет получить доступ к значению заголовка HTTP по ключу. Например, для просмотра типа содержимого ответного пейлоада, требуется использовать Content-Type.

```
>>> response.headers['Content-Type']
'application/json; charset=utf-8'
```

У объектов словарей в качестве заголовков есть своим особенности. Специфика HTTP предполагает, что заголовки не чувствительны к регистру. Это значит, что при получении доступа к заголовкам можно не беспокоится о том, использованы строчным или прописные буквы.

При использовании ключей 'content-type' и 'Content-Type' результат будет получен один и тот же.

Python Requests параметры запроса

Наиболее простым способом настроить запрос GET является передача значений через параметры строки запроса в URL. При использовании метода get(), данные передаются в params. Например, для того, чтобы посмотреть на библиотеку requests можно использовать Search API на GitHub.

```
import requests

# Поиск местонахождения для запросов на GitHub
response = requests.get(
    'https://api.github.com/search/repositories',
    params={'q': 'requests+language:python'},
)

# Анализ некоторых атрибутов местонахождения запросов
json_response = response.json()
repository = json_response['items'][0]
print(f'Repository name: {repository["name"]}')  # Python 3.6+
print(f'Repository description: {repository["description"]}')  # Python 3.6+
```

Передавая словарь {'q': 'requests+language:python'} в параметр params, который является частью .get(), можно изменить ответ, что был получен при использовании Search API.

Можно передать параметры в get() в форме словаря, как было показано выше. Также можно использовать список кортежей.

```
>>> requests.get(
...     'https://api.github.com/search/repositories',
...     params=[('q', 'requests+language:python')],
... )
<Response [200]>
```

Также можно передать значение в байтах.

```
>>> requests.get(
...     'https://api.github.com/search/repositories',
...     params=b'q=requests+language:python',
... )
<Response [200]>
```

Строки запроса полезны для уточнения параметров в запросах GET. Также можно настроить запросы при помощи добавления или изменения заголовков отправленных сообщений.

Настройка HTTP заголовка запроса (headers)

Для изменения HTTP заголовка требуется передать словарь данного HTTP заголовка в get() при помощи использования параметра headers. Например, можно изменить предыдущий поисковой запрос, подсветив совпадения в результате. Для этого в заголовке Accept медиа тип уточняется при помощи text-match.

```
import requests

response = requests.get(
    'https://api.github.com/search/repositories',
    params={'q': 'requests+language:python'},
    headers={'Accept': 'application/vnd.github.v3.text-match+json'},
)

# просмотр нового массива `text-matches` с предоставленными данными
# о поиске в пределах результатов
json_response = response.json()
repository = json_response['items'][0]
print(f'Text matches: {repository["text_matches"]}')
```

Заголовок Accept сообщает серверу о типах контента, который можно использовать в рассматриваемом приложении. Здесь подразумевается, что все совпадения будут подсвечены, для чего в заголовке используется значение application/vnd.github.v3.text-match+json. Это уникальный заголовок Accept для GitHub. В данном случае содержимое представлено в специальном JSON формате.

Примеры HTTP методов в Requests

Помимо GET, большой популярностью пользуются такие методы, как POST, PUT, DELETE, HEAD, PATCH и OPTIONS. Для каждого из этих методов существует своя сигнатура, которая очень похожа на метод get().

```
>>> requests.post('https://httpbin.org/post', data={'key':'value'})

>>> requests.put('https://httpbin.org/put', data={'key':'value'})

>>> requests.delete('https://httpbin.org/delete')

>>> requests.head('https://httpbin.org/get')

>>> requests.patch('https://httpbin.org/patch', data={'key':'value'})

>>> requests.options('https://httpbin.org/get')
```

Каждая функция создает запрос к httpbin сервису, используя при этом ответный HTTP метод.

```
>>> response = requests.head('https://httpbin.org/get')
>>> response.headers['Content-Type']
'application/json'

>>> response = requests.delete('https://httpbin.org/delete')
>>> json_response = response.json()
>>> json_response['args']
{}
```

При использовании каждого из данных методов в Response могут быть возвращены заголовки, тело запроса, коды состояния и многие другие аспекты. Методы POST, PUT и PATCH в дальнейшем будут описаны более подробно.

Python Requests анализ запроса

При составлении запроса стоит иметь в виду, что перед его фактической отправкой на целевой сервер библиотека `requests` выполняет определенную подготовку. Подготовка запроса включает в себя такие вещи, как **проверка заголовков** и **сериализация содержимого JSON**.

Если открыть `.request`, можно просмотреть `PreparedRequest`.

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

>>> response.request.url
'https://httpbin.org/post'

>>> response.request.body
b'{"key": "value"}'
```

Проверка PreparedRequest открывает доступ ко всей информации о выполняемом запросе. Это может быть пейлоад, URL, заголовки, аутентификация и многое другое.
У всех описанных ранее типов запросов была одна общая черта – они представляли собой неаутентифицированные запросы к публичным API. Однако, подобающее большинство служб, с которыми может столкнуться пользователь, запрашивают аутентификацию.

Python Requests аутентификация HTTP AUTH

Аутентификация помогает сервису понять, кто вы. Как правило, вы предоставляете свои учетные данные на сервер, передавая данные через заголовок `Authorization` или пользовательский заголовок, определенной службы. Все функции запроса, которые вы видели до этого момента, предоставляют параметр с именем `auth`, который позволяет вам передавать свои учетные данные.

Одним из примеров API, который требует аутентификации, является Authenticated User API на GitHub. Это конечная точка веб-сервиса, которая предоставляет информацию о профиле аутентифицированного пользователя. Чтобы отправить запрос API-интерфейсу аутентифицированного пользователя, вы можете передать свое имя пользователя и пароль на GitHub через кортеж в `get()`.

```
>>> from getpass import getpass
>>> requests.get('https://api.github.com/user', auth=('username', getpass()))
<Response [200]>
```

Запрос выполнен успешно, если учетные данные, которые вы передали в кортеже `auth`, действительны. Если вы попытаетесь сделать этот запрос без учетных данных, вы увидите, что код состояния `401 Unauthorized`.

```
>>> requests.get('https://api.github.com/user')
<Response [401]>
```

Python Requests производительность приложений

При использовании `requests`, особенно в среде приложений, важно учитывать влияние на производительность. Такие функции, как **контроль таймаута**, сеансы и ограничения повторных попыток, могут помочь обеспечить бесперебойную работу приложения.

Таймауты

Когда вы отправляете встроенный запрос во внешнюю службу, вашей системе нужно будет дождаться ответа, прежде чем двигаться дальше. Если ваше приложение слишком **долго ожидает ответа**, запросы к службе могут быть сохранены, пользовательский интерфейс может пострадать или фоновые задания могут зависнуть.

По умолчанию в requests на ответ время не ограничено, и весь процесс может занять значительный промежуток. По этой причине вы всегда должны **указывать время ожидания**, чтобы такого не происходило. Чтобы **установить время ожидания запроса**, используйте параметр `timeout`. `timeout` может быть целым числом или числом с плавающей точкой, представляющим количество секунд ожидания ответа до истечения времени ожидания.

```
>>> requests.get('https://api.github.com', timeout=1)
<Response [200]>
>>> requests.get('https://api.github.com', timeout=3.05)
<Response [200]>
```

Вы также можете передать кортеж. Это – таймаут соединения (время, за которое клиент может установить соединение с сервером), а второй – таймаут чтения (время ожидания ответа, как только ваш клиент установил соединение):

```
>>> requests.get('https://api.github.com', timeout=(2, 5))
<Response [200]>
```