

# Подробное введение в Python часть 6

## Requests в Python

### 1. Введение API и JSON

Библиотека requests является стандартным инструментом для составления HTTP-запросов в Python. Простой и аккуратный [API](#) значительно облегчает трудоемкий процесс создания запросов. Таким образом, можно сосредоточиться на взаимодействии со службами и использовании данных в приложении.

Аббревиатура [API](#) соответствует английскому application programming interface — программный интерфейс приложения. По сути, API действует как коммуникационный уровень или интерфейс, который позволяет различным системам взаимодействовать друг с другом без необходимости точно понимать, что делает каждая из систем.

[API](#)-интерфейсы имеют разные формы. Это может быть API операционной системы, используемый для включения камеры и микрофона для присоединения к звонку Zoom. Или это могут быть веб-API, используемые для действий, ориентированных на веб, таких как лайки фотографий в Instagram или получение последних твитов.

Независимо от типа, все API-интерфейсы работают приблизительно одинаково. Обычно программа-клиент запрашивает информацию или данные, а API возвращает ответ в соответствии с тем, что мы запросили. Каждый раз, когда мы открываем Twitter или прокручиваем ленту Instagram, приложение делает запрос к API и просто отображает ответ с учетом дизайна программы.

В это лекции мы подробно остановимся на высокоуровневых веб-API, которые обмениваются информацией между сетями.

### SOAP vs REST vs GraphQL

В конце 1990-х и начале 2000-х годов две разные модели дизайна API стали нормой для публичного доступа к данным:

SOAP (Simple Object Access Protocol) ассоциируется с корпоративным миром, имеет строгую систему на основе «контрактов». Этот подход в основном связан скорее с обработкой действий, чем с данными.

REST (Representational State Transfer) используется для общедоступных API и идеально подходит для получения данных из интернета.

Сегодня распространение также получает GraphQL — созданный Facebook гибкий язык API-запросов. Хотя GraphQL находится на подъеме и внедряется крупными компаниями, включая GitHub и Shopify, большинство общедоступных API-интерфейсов это REST API. Поэтому в рамках лекции

мы ограничимся именно REST-подходом и тем, как взаимодействовать с такими API с помощью Python.

## requests и API

При использовании API с Python нам понадобится всего одна библиотека: `requests`. С её помощью вы сможете выполнять бóльшую часть, если не все, действия, необходимые для использования любого общедоступного API.

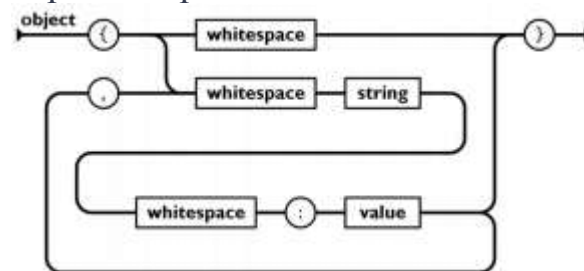
## JSON

JSON – текстовый формат данных, используемый практически во всех скриптовых языках программирования, однако его истоки находятся у JavaScript. Он имеет сходство с буквенным синтаксисом данного языка программирования, но может использоваться отдельно от него. Многие среды разработки отлично справляются с его чтением и генерированием. JSON находится в состоянии строки, поэтому позволяет передавать информацию по сети. Он преобразуется в объект JS, чтобы пользователь мог прочитать эти данные. Осуществляется это методами языка программирования, но сам JSON методов не имеет, только свойства.

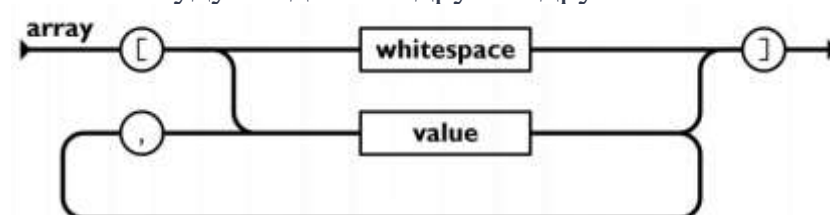
Вы можете сохранить текстовый файл JSON в собственном формате *.json*, и он будет отображаться как текстовый. Для MIME Type представление меняется на *application/json*.

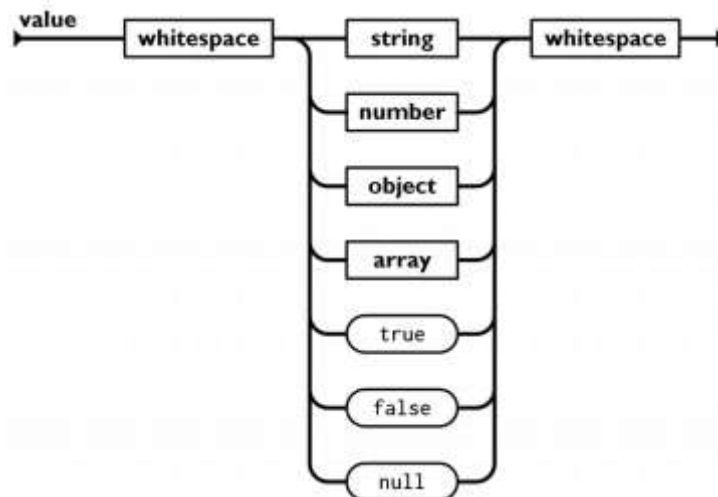
## Структура JSON

При работе с рассматриваемым текстовым форматом необходимо учитывать правила создания его структуры в объекте, массиве и при присвоении значения. На следующей иллюстрации вы видите наглядную демонстрацию представления объекта.



Если речь идет о массиве, здесь тоже необходимо применять определенные правила, поскольку он всегда представляет собой упорядоченную совокупность данных и находится внутри скобок [ ]. При этом значения будут отделены друг от друга.





Если вам интересно, на официальном сайте JSON можно найти более детальное описание всех значений и использования формата в разных языках программирования со списком всех доступных библиотек и инструментов.

#### Основные преимущества JSON

Как уже понятно, JSON используется для обмена данными, которые являются структурированными и хранятся в файле или в строке кода. Числа, строки или любые другие объекты отображаются в виде текста, поэтому пользователь обеспечивает простое и надежное хранение информации. JSON обладает рядом преимуществ, которые и сделали его популярным:

- Не занимает много места, является компактным в написании и быстро компилируется.
- Создание текстового содержимого понятно человеку, просто в реализации, а чтение со стороны среды разработки не вызывает никаких проблем. Чтение может осуществляться и человеком, поскольку ничего сложного в представлении данных нет.
- Структура преобразуется для чтения на любых языках программирования.
- Практически все языки имеют соответствующие библиотеки или другие инструменты для чтения данных JSON.

#### Основной принцип работы JSON

Разберемся, в чем состоит основной принцип работы данного формата, где он используется и чем может быть полезен для обычного пользователя и разработчика.

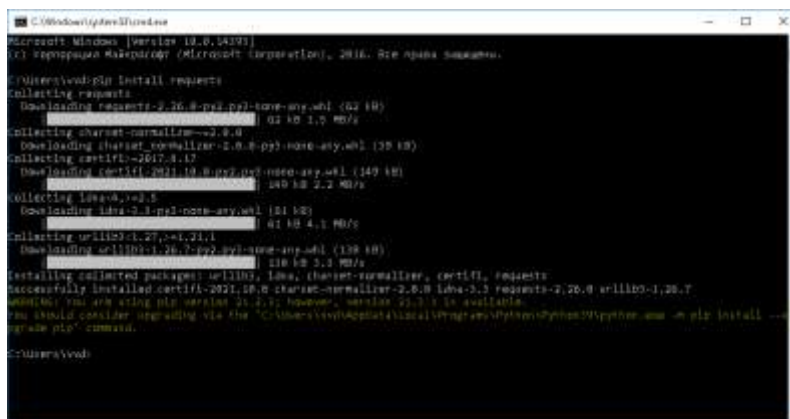
Ниже приведена примерная структура обработки данных при обращении «клиент-сервер-клиент». Это актуально для передачи информации с сервера в браузер по запросу пользователя, что и является основным предназначением JSON.

- а) Запрос на сервер отправляется по клику пользователя, например, когда он открывает элемент описания чего-либо для его детального прочтения.

- b) Запрос генерируется при помощи AJAX с использованием JavaScript и программного сценарного файла PHP. Сам сценарий запущен на сервере, значит, поиск данных завершится успешно.
- c) Программный файл PHP запоминает всю предоставленную с сервера информацию в виде строки кода.
- d) JavaScript берет эту строку, восстанавливает ее до необходимого состояния и выводит информацию на странице пользователя в браузере.
- e) На выполнение этой задачи понадобится меньше секунды, и главную роль здесь выполняет встроенный в браузер JavaScript. Если же он по каким-то причинам не функционирует или отсутствует, действие произведено не будет.

## 1.1 Python установка библиотеки requests

Установка для windows в командной строке `pip install requests`



```
C:\Windows\system32\cmd.exe
Microsoft Windows [version 10.0.14393]
(c) 2016 Microsoft Corporation. Все права защищены.

C:\Users\Yend>pip install requests
Collecting requests
  Downloading requests-2.16.4-py2.py3-none-any.whl (62 kB)
    |#####| 62 kB 1.5 MB/s
Collecting charset-normalizer<2.0.0
  Downloading charset-normalizer-2.0.0-py2.py3-none-any.whl (39 kB)
Collecting certifi<2021.4.12
  Downloading certifi-2021.10.8-py2.py3-none-any.whl (149 kB)
    |#####| 149 kB 2.2 MB/s
Collecting idna<3.0
  Downloading idna-3.0-py2.py3-none-any.whl (61 kB)
    |#####| 61 kB 4.1 MB/s
Collecting urllib3<1.27,>=1.21.1
  Downloading urllib3-1.26.7-py2.py3-none-any.whl (139 kB)
    |#####| 139 kB 5.3 MB/s
Installing collected packages: urllib3, idna, charset-normalizer, certifi, requests
Successfully installed certifi-2021.10.8 charset-normalizer-2.0.0 idna-3.0 requests-2.26.4 urllib3-1.26.7
WARNING: You are using pip version 21.1.3, however, version 22.1 is available.
You should consider upgrading via the 'C:\Users\Yend\AppData\Local\Programs\Python\Python39\python.exe -m pip install --
upgrade pip' command.

C:\Users\Yend>
```

Сразу после установки requests можно полноценно использовать в приложении. Импорт requests производится следующим образом.

```
import requests
```

Таким образом, все подготовительные этапы для последующего использования requests завершены.

## 1.2 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]>
```

### 1.3 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.

### 1.4 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.')
```

В таком случае, если с сервера будет получен код состояния 200, тогда программа выведет значение `Success!`. Однако, если от сервера поступит код 404, тогда программа выведет значение `Not Found`. `requests` может значительно упростить весь процесс. 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.

Анализ способов использования кодов состояния, полученных с сервера, является неплохим стартом для изучения `requests`. Тем не менее, при создании **запроса GET**, значение кода состояния является не самой важной информацией, которую хочет получить программист. Обычно запрос производится для извлечения более содержательной информации. В дальнейшем будет показано, как добраться до актуальных данных, которые сервер высылает отправителю в ответ на запрос.

## 1.5 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
b'{"current_user_url":"https://api.github.com/user","current_user_a
uthorizations_html_url":.....'
```

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
'{"current_user_url":"https://api.github.com/user","current_user_au
thorizations_html_url":"https://github.com/....."
```

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
'{"current_user_url":"https://api.github.com/user","current_user_au
thorizations_html_url":"https://github.com/settings/connections/app
lications{/client_id}","authorizations_url":"https://api.github.com
/authorizations","code_search_url":"https://api.github.com/search/c
ode?q={query}{&page,per_page,sort,order}","commit_search_url":"http
s://api.github.com/search/commits?q={query}{&page,per_page,sort,ord
er}","emails_url":"https://api.github.com/user/emails","emojis_url"
:"https://api.github.com/emojis","events_url":"https://api.github.c
om/events","feeds_url":"https://api.github.com/feeds","followers_ur
l":"https://api.github.com/user/followers","following_url":"https:/
/api.github.com/user/following{/target}","gists_url":"https://api.g
ithub.com/gists{/gist_id}","hub_url":"https://api.github.com/hub","
issue_search_url":"https://api.github.com/search/issues?q={query}{&
page,per_page,sort,order}","issues_url":"https://api.github.com/iss
ues","keys_url":"https://api.github.com/user/keys","notifications_u
rl":"https://api.github.com/notifications","organization_repositori
es_url":"https://api.github.com/orgs/{org}/repos{?type,page,per_pag
e,sort}","organization_url":"https://api.github.com/orgs/{org}","pu
blic_gists_url":"https://api.github.com/gists/public","rate_limit_u
rl":"https://api.github.com/rate_limit","repository_url":"https://a
pi.github.com/repos/{owner}/{repo}","repository_search_url":"https:
//api.github.com/search/repositories?q={query}{&page,per_page,sort,
order}","current_user_repositories_url":"https://api.github.com/use
r/repos{?type,page,per_page,sort}","starred_url":"https://api.githu
b.com/user/starred{/owner}/{repo}","starred_gists_url":"https://api
.github.com/gists/starred","team_url":"https://api.github.com/teams
","user_url":"https://api.github.com/users/{user}","user_organizati
ons_url":"https://api.github.com/user/orgs","user_repositories_url"
:"https://api.github.com/users/{user}/repos{?type,page,per_page,sor
t}","user_search_url":"https://api.github.com/search/users?q={query}
{&page,per_page,sort,order}"}'
```

Если присмотреться к ответу, можно заметить, что его содержимое является сериализованным **JSON** контентом. Воспользовавшись словарем, можно взять полученные из .text строки str и провести с ними обратную сериализацию при помощи использования json.loads(). Есть и более простой способ, который требует применения json().

```
>>> response.json()
{'current_user_url': 'https://api.github.com/user',
 'current_user_authorizations_html_url':
 'https://github.com/settings/connections/applications{/client_id}',
 'authorizations_url': 'https://api.github.com/authorizations',
 'code_search_url':
 'https://api.github.com/search/code?q={query}{&page,per_page,sort,o
rder}',
 'commit_search_url':
 'https://api.github.com/search/commits?q={query}{&page,per_page,sor
t,order}',
 'emails_url': 'https://api.github.com/user/emails',
 'emojis_url': 'https://api.github.com/emojis',
 'events_url': 'https://api.github.com/events',
 'feeds_url': 'https://api.github.com/feeds',
 'followers_url':
```



```
'https://api.github.com/user/followers', 'following_url':
'https://api.github.com/user/following{/target}', 'gists_url':
'https://api.github.com/gists{/gist_id}', 'hub_url':
'https://api.github.com/hub', 'issue_search_url':
'https://api.github.com/search/issues?q={query}{&page,per_page,sort,order}', 'issues_url': 'https://api.github.com/issues',
'keys_url': 'https://api.github.com/user/keys',
'notifications_url': 'https://api.github.com/notifications',
'organization_repositories_url':
'https://api.github.com/orgs/{org}/repos{?type,page,per_page,sort}',
'organization_url': 'https://api.github.com/orgs/{org}',
'public_gists_url': 'https://api.github.com/gists/public',
'rate_limit_url': 'https://api.github.com/rate_limit',
'repository_url': 'https://api.github.com/repos/{owner}/{repo}',
'repository_search_url':
'https://api.github.com/search/repositories?q={query}{&page,per_page,sort,order}', 'current_user_repositories_url':
'https://api.github.com/user/repos{?type,page,per_page,sort}',
'starred_url':
'https://api.github.com/user/starred{/owner}/{repo}',
'starred_gists_url': 'https://api.github.com/gists/starred',
'team_url': 'https://api.github.com/teams', 'user_url':
'https://api.github.com/users/{user}', 'user_organizations_url':
'https://api.github.com/user/orgs', 'user_repositories_url':
'https://api.github.com/users/{user}/repos{?type,page,per_page,sort}', 'user_search_url':
'https://api.github.com/search/users?q={query}{&page,per_page,sort,order}'} }
```

## 1.6 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
{'Server': 'GitHub.com', 'Date': 'Mon, 10 Dec 2018 17:49:54 GMT',
'Content-Type': 'application/json; charset=utf-8', 'Transfer-
Encoding': 'chunked', 'Status': '200 OK', 'X-RateLimit-Limit':
'60', 'X-RateLimit-Remaining': '59', 'X-RateLimit-Reset':
'1544467794', 'Cache-Control': 'public, max-age=60, s-maxage=60',
'Vary': 'Accept', 'ETag': 'W/"7dc470913f1fe9bb6c7355b50a0737bc"',
'X-GitHub-Media-Type': 'github.v3; format=json', 'Access-Control-
Expose-Headers': 'ETag, Link, Location, Retry-After, X-GitHub-OTP,
X-RateLimit-Limit, X-RateLimit-Remaining, X-RateLimit-Reset, X-
OAuth-Scopes, X-Accepted-OAuth-Scopes, X-Poll-Interval, X-GitHub-
Media-Type', 'Access-Control-Allow-Origin': '*', 'Strict-Transport-
Security': 'max-age=31536000; includeSubdomains; preload', 'X-
Frame-Options': 'deny', 'X-Content-Type-Options': 'nosniff', 'X-
XSS-Protection': '1; mode=block', 'Referrer-Policy': 'origin-when-
cross-origin, strict-origin-when-cross-origin', 'Content-Security-
Policy': "default-src 'none'", 'Content-Encoding': 'gzip', 'X-
GitHub-Request-Id': 'E439:4581:CF2351:1CA3E06:5C0EA741'}
```

.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'
```

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

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

Это была основная информация, требуемая для работы с Response. Были задействованы главные атрибуты и методы, а также представлены примеры их использования. Далее в лекции будет показано, как изменится ответ после настройки запроса GET.

## 2. 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.

## 2.1 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 Ассерпт 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 Ассерпт 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 Ассерпт для GitHub. В данном случае содержимое представлено в специальном **JSON формате**.

Перед более глубоким изучением способов редактирования запросов, будет не лишним остановиться на некоторых других методах HTTP.

## 2.2 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. Результат каждого метода можно изучить способом, который был использован в предыдущих примерах.

```
>>> 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.

## 2.3 Python Requests тело сообщения

В соответствии со спецификацией HTTP запросы POST, PUT и PATCH передают информацию через тело сообщения, а не через параметры строки запроса. Используя requests, можно передать данные в параметр data.

В свою очередь data использует **словарь**, список кортежей, байтов или объект файла. Это особенно важно, так как может возникнуть необходимость адаптации отправляемых с запросом данных в соответствии с определенными параметрами сервера.

К примеру, если тип содержимого запроса application/x-www-form-urlencoded, можно отправить данные формы в виде словаря.

```
>>> requests.post('https://httpbin.org/post',
data={'key':'value'})
<Response [200]>
```

Ту же самую информацию также можно отправить в виде списка кортежей.

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

```
<Response [200]>
```

В том случае, если требуется отправить данные JSON, можно использовать параметр `json`. При передачи данных JSON через `json`, `requests` произведет сериализацию данных и добавит правильный `Content-Type` заголовок.

Стоит взять на заметку сайт [httpbin.org](http://httpbin.org). Это чрезвычайно полезный ресурс, созданный человеком, который внедрил использование `requests` – Кеннетом Рейтцом. Данный сервис предназначен для тестовых запросов. Здесь можно составить пробный запрос и получить ответ с требуемой информацией. В качестве примера рассмотрим базовый запрос с использованием POST.

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

Здесь видно, что сервер получил данные и HTTP заголовки, отправленные вместе с запросом. `requests` также предоставляет информацию в форме `PreparedRequest`.

## 2.4 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 посмотреть `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.

## 2.5 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]>
```

Когда вы передаете имя пользователя и пароль в кортеже параметру auth, вы используете учетные данные при помощи базовой схемы аутентификации HTTP.

Таким образом, вы можете создать тот же запрос, передав подробные учетные данные базовой аутентификации, используя HTTPBasicAuth.

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

Хотя вам не нужно явно указывать обычную аутентификацию, может потребоваться аутентификация с использованием другого метода. Requests предоставляет другие методы аутентификации, например, HTTPDigestAuth и HTTPProxyAuth.

Вы даже можете предоставить свой собственный механизм аутентификации. Для этого необходимо сначала создать подкласс **AuthBase**. Затем происходит имплементация `__call__()`.



```
import requests
from requests.auth import AuthBase

class TokenAuth(AuthBase):
    """Implements a custom authentication scheme."""

    def __init__(self, token):
        self.token = token

    def __call__(self, r):
        """Attach an API token to a custom auth header."""
        r.headers['X-TokenAuth'] = f'{self.token}' # Python 3.6+
        return r

requests.get('https://httpbin.org/get',
auth=TokenAuth('12345abcde-token'))
```

Здесь пользовательский механизм `TokenAuth` получает специальный токен. Затем этот токен включается заголовок `X-TokenAuth` запроса.

## 2.6 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]>
```

В первом примере запрос истекает через 1 секунду. Во втором примере запрос истекает через 3,05 секунды.

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]>
```

Если запрос устанавливает соединение в течение 2 секунд и получает данные в течение 5 секунд после установления соединения, то ответ будет возвращен, как это было раньше. Если время ожидания истекло, функция вызовет исключение `Timeout`.

```
import requests  
from requests.exceptions import Timeout  
  
try:  
    response = requests.get('https://api.github.com', timeout=1)  
except Timeout:  
    print('The request timed out')  
else:  
    print('The request did not time out')
```

Ваша программа может поймать исключение `Timeout` и ответить соответственно.

## 2.7 Объект `Session` в `Requests`

До сих пор вы имели дело с `requests API` высокого уровня, такими как `get()` и `post()`. Эти функции являются абстракцией того, что происходит, когда вы делаете свои запросы. Они скрывают детали реализации, такие как управление соединениями, так что вам не нужно о них беспокоиться.

Под этими абстракциями находится класс под названием `Session`. Если вам необходимо настроить контроль над выполнением запросов или повысить производительность ваших запросов, вам может потребоваться использовать `Session` напрямую.

Сессии используются для сохранения параметров в запросах.

Например, если вы хотите использовать одну и ту же аутентификацию для нескольких запросов, вы можете использовать **сеанс**:

```
import requests  
from getpass import getpass  
  
# используя менеджер контента, можно убедиться, что ресурсы,  
# применимые  
# во время сессии будут свободны после использования  
with requests.Session() as session:  
    session.auth = ('username', getpass())  
  
    # Instead of requests.get(), you'll use session.get()  
    response = session.get('https://api.github.com/user')
```

```
# здесь можно изучить ответ
print(response.headers)
print(response.json())
```

Каждый раз, когда вы делаете запрос `session`, после того как он был инициализирован с учетными данными аутентификации, учетные данные будут сохраняться.

Первичная оптимизация производительности сеансов происходит в форме постоянных соединений. Когда ваше приложение устанавливает соединение с сервером с помощью `Session`, оно сохраняет это соединение в пуле соединений. Когда ваше приложение снова хочет подключиться к тому же серверу, оно будет использовать соединение из пула, а не устанавливать новое.

## 2.8 HTTPAdapter — Максимальное количество повторов запроса в Requests

В случае сбоя запроса возникает необходимость сделать повторный запрос. Однако `requests` не будет делать это самостоятельно. Для применения функции повторного запроса требуется реализовать собственный транспортный адаптер.

Транспортные адаптеры позволяют определить набор конфигураций для каждой службы, с которой вы взаимодействуете. Предположим, вы хотите, чтобы все запросы к `https://api.github.com` были повторены три раза, прежде чем, наконец, появится `ConnectionError`. Для этого нужно построить транспортный адаптер, установить его параметр `max_retries` и подключить его к существующему объекту `Session`.

```
import requests
from requests.adapters import HTTPAdapter
from requests.exceptions import ConnectionError

github_adapter = HTTPAdapter(max_retries=3)

session = requests.Session()

# использование `github_adapter` для всех запросов, которые
# начинаются с указанным URL
session.mount('https://api.github.com', github_adapter)

try:
    session.get('https://api.github.com')
except ConnectionError as ce:
    print(ce)
```

При установке `HTTPAdapter`, `github_adapter` к `session`, `session` будет придерживаться своей конфигурации для каждого запроса к `https://api.github.com`.

Таймауты, транспортные адаптеры и сессии предназначены для обеспечения эффективности используемого кода и **стабильности приложения**.

### ИТОГ

Изучение библиотеки **Python requests** является очень трудоемким процессом. Грамотное использование **requests** позволит наиболее эффективно настроить разрабатываемые приложения, исследуя широкий спектр веб-сервисов и данных, опубликованных на них.