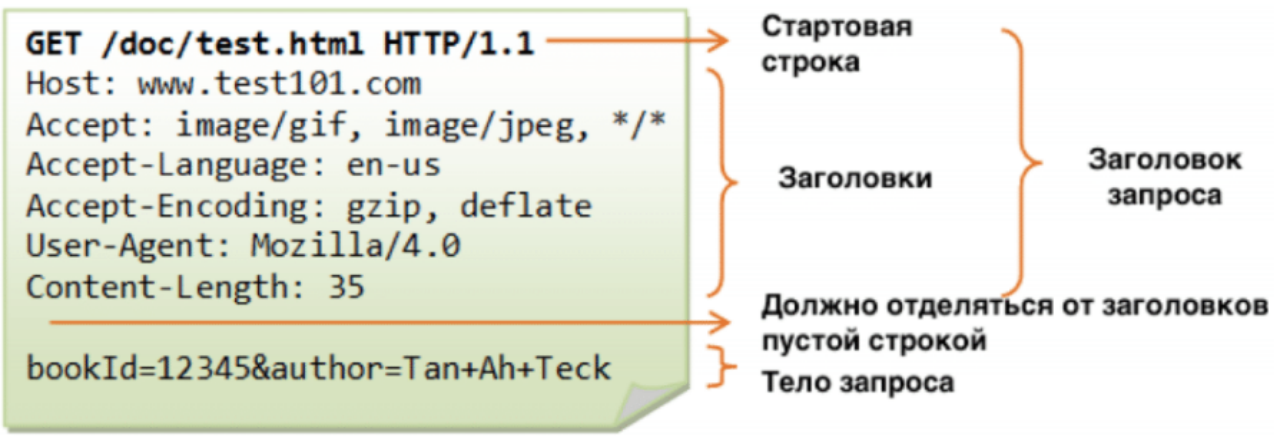
URI

Единообразный идентификатор ресурса URI (Uniform Resource Identifer) представляет собой короткую последовательность символов, идентифицирующую абстрактный или физический ресурс. URI не указывает на то, как получить ресурс, а только идентифицирует его. Один из самых известных примеров URI — это URL.
Для однозначной идентификации ресурсов в сети Веб используются уникальные идентификаторы URL (Uniform Resource Locator). Имеет следующую структуру:<схема>://<логин>:<пароль>@<хост>: <порт>/<URL-путь>

Структура HTTP-запроса

Теперь рассмотрим структуру HTTP-запроса.



Совокупность стартовой строки и заголовков называют заголовком запроса. После заголовка запроса идет пустая строка, затем идёт тело запроса.
Стартовая строка является обязательным элементом, так как указывает на тип запроса/ответа, заголовки и тело сообщения могут отсутствовать. Стартовые строки различаются для запроса и ответа. Строка запроса выглядит так: Метод Путь HTTP/Версия протокола.
Методы протокола:

- GET: используется для получения какой-то информации;
- HEAD: аналогично GET-запросу, но без тела запроса;
- POST: отсылает от клиента данные на сервер;
- PATCH: изменять отправленные данные;

- DELETE: удалить данные.
- ...

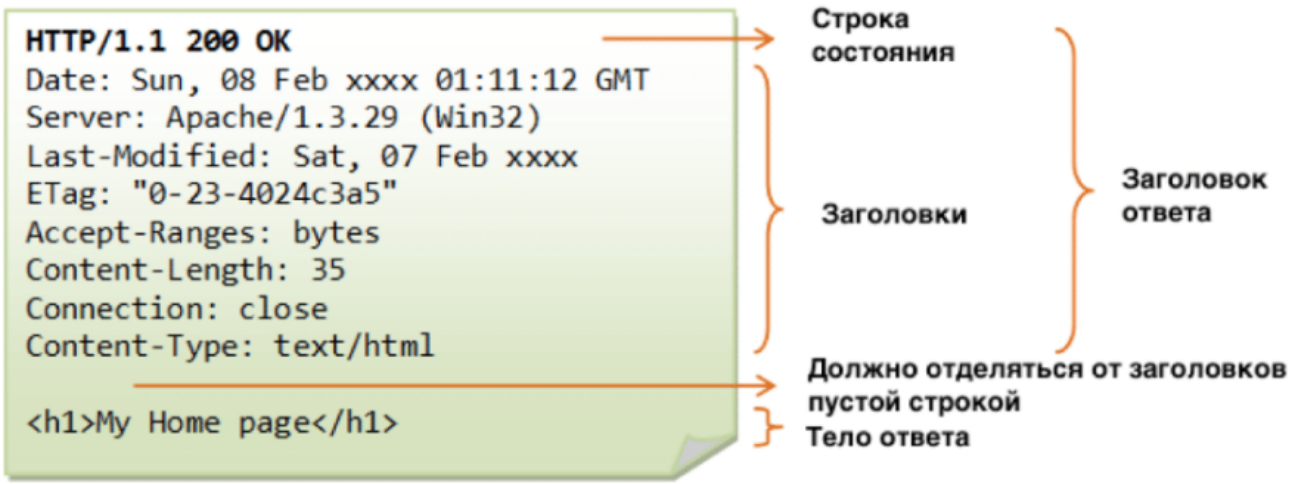
Поля заголовка, следующие за строкой состояния, позволяют уточнять запрос, т.е. передавать серверу дополнительную информацию. Поле заголовка имеет следующий формат: Имя_поля: Значение. Поля заголовка:

- Host — доменное имя или IP-адрес узла, к которому обращается клиент;
- Referer — URL, откуда перешел клиент;
- Accept — MIME-типы данных, обрабатываемых клиентом;
- Accept-Charset — перечень поддерживаемых кодировок;
- Content-Type — MIME-тип данных, содержащихся в теле запроса;
- Content-Length — число символов, содержащихся в теле запроса;
- Connection — используется для управления TCP-соединением;
- User-Agent — информация о клиенте.

MIME

Спецификация MIME (Multipurpose Internet Mail Extension — многоцелевое почтовоерасширение Internet) первоначально была разработана для того, чтобы обеспечить передачу различных форматов данных в составе электронных писем. До появления MIME устройства, взаимодействующие по протоколу HTTP, обменивались исключительно текстовой информацией. Для описания формата данных используются тип и подтип. Тип определяет, к какому классу относится формат содержимого HTTP-запроса или HTTP-ответа. Подтип уточняет формат (text/html, image/png). Данный спецификатор позволяет передавать от клиента к серверу различные типы данных.

Структура HTTP-ответа



В совокупности строка состояния и заголовки называются заголовком ответа. После этого идет пустая строка, после пустой строки идет тело ответа. Строка состояния состоит из версии протокола и статуса HTTP-ответа и расшифровки этого ответа. Существует 5 классов ответов:

- 1xx — специальный класс сообщений, называемых информационными. Означает, что сервер продолжает обработку запроса. (101 Switching Protocols);
- 2xx — успешная обработка запроса клиента. (200 Ok, 201 Created);
- 3xx — перенаправление запроса. (301 Moved Permanently, 302 Found);
- 4xx — ошибка клиента. (400 Bad Request, 403 Forbidden, 404 Not Found);
- 5xx — ошибка сервера. (500 Internal Server Error, 502 Bad Gateway).

Рассмотрим поля заголовка ответа:

- Server — имя и номер версии сервера;
- Allow — список методов, допустимых для данного ресурса;
- Content-Type — MIME-тип данных, содержащихся в теле ответа сервера;
- Content-Length — число символов, содержащихся в теле ответа сервера;
- Last-Modifed — дата и время последнего изменения ресурса;

- Expires — дата и время, когда информация станет устаревшей;
- Location — расположение ресурса;
- Cache-Control — директива управления кэшированием.

Работа с HTTP из python

Библиотека requests

Импортируем библиотеку и попробуем выполнить простой get-запрос.

```
import requests

# Make a Request

r = requests.get('http://httpbin.org/get')
print(r.text)
```

В итоге мы получили HTTP-ответ со следующей структурой:


```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },
  "origin": "185.136.79.217",
  "url": "http://httpbin.org/get"
}
```


Допустим, мы хотим выполнить post-запрос.

```
r = requests.post('http://httpbin.org/post')
print(r.text)
```


```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {},

  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Content-Length": "0",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },
  "json": null,
  "origin": "185.136.79.217",
  "url": "http://httpbin.org/post"

}
```


Чтобы передать Get-параметры, нам необходимо указать params.

```python
# Passing Parameters
payload = {'key1': 'value1', 'key2': 'value2'}
r = requests.get('http://httpbin.org/get', params=payload)
print(r.text)
```

```
{
  "args": {
    "key1": "value1",
    "key2": "value2"
  },

  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },

  "origin": "185.136.79.217",
  "url": "http://httpbin.org/get?key1=value1&key2=value2"
}
```

Чтобы передать данные в Post Put Patch запросах, нам нужно передать в соответствующие методы поля data.

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


```
{
  "args": {},
  "data": "",
  "files": {},
  "form": {
    "key": "value"
  },

  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Content-Length": "9",
    "Content-Type": "application/x-www-form-urlencoded",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },

  "json": null,
  "origin": "185.136.79.217",
  "url": "http://httpbin.org/put"
}
```


Если мы хотим передать json, это можно сделать двумя путями: руками реализовать json и передать в поле data, либо использовть встроенный в библиотеку механизм и передать поле json, которое сделает все само.

```
import json
url = 'http://httpbin.org/post'
r = requests.post(url, data=json.dumps({'key':'value'}))
r = requests.post(url, json={'key':'value'})
print(r.text)
```


```
{
  "args": {},
  "data": "{\"key\": \"value\"}",
  "files": {},
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Content-Length": "16",
    "Content-Type": "application/json",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"

  },

  "json": {
    "key": "value"
  },

  "origin": "185.136.79.217",
  "url": "http://httpbin.org/post"
}
```


В поле data пришел реализованный json, а в поле json отобразились ключ и значение.

Рассмотрим, как передать файл на сервер. Для этого необходимо объявить переменную files, которая будет являться словарём, где ключ – это ключ, по которому сможем получить файл на сервере. Дальше идет имя файла и его дескриптор. Потом эту переменную необходимо передать в поле files.

```python
# POST a Multipart-Encoded File
url = 'http://httpbin.org/post'
files = {'file':
         ('test.txt',
          open('/Users/alexander/Desktop/test.txt',
               'rb'))}

r = requests.post(url, files=files)
print(r.text)
```


```
{
  "args": {},
  "data": "",
  "files": {
    "file": "test content\n"
  },
  "form": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Content-Length": "157",
    "Content-Type": "multipart/form-data; boundary=a6d397e696144b588e9a4aa1cff723fb",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },
  "json": null,
  "origin": "185.136.79.217",
  "url": "http://httpbin.org/post"
}
```


Рассмотрим, как передать заголовки.

```python
# Headers
url = 'http://httpbin.org/get'
headers = {'user-agent': 'my-app/0.0.1'}

r = requests.get(url, headers=headers)
print(r.text)
```

```
{
  "args": {},

  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "my-app/0.0.1"

  },
  "origin": "185.136.79.217",
  "url": "http://httpbin.org/get"

}
```

Мы посмотрели, как же нам передать различные параметры в запрос, теперь давайте рассмотрим ответы. Существует несколько способов получения HTTP-ответа: текст, контент и json. Текст вернет данные с типом string, контент – массив байт, json – словарь.

```python
# Response Content
r = requests.get('http://httpbin.org/get')
print(type(r.text), r.text)
print(type(r.content), r.content)
print(type(r.json()), r.json())
```

```
{

<class 'str'> {
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Accept-Encoding": "gzip, deflate",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "python-requests/2.18.1"
  },
  "origin": "185.136.79.217",
  "url": "http://httpbin.org/get"
}

<class 'bytes'> b'{\n  "args": {}, \n  "headers": {\n    "Accept": "*/*", \n
"Accept-Encoding":
"gzip, deflate", \n    "Connection": "close", \n    "Host": "httpbin.org", \n
"User-Agent":
"python-requests/2.18.1"\n  }, \n  "origin": "185.136.79.217", \n  "url":
"http://httpbin.org/get"\n}\n'
<class 'dict'> {'args': {}, 'headers': {'Accept': '*/*', 'Accept-Encoding': 'gzip,
deflate',
'Connection': 'close', 'Host': 'httpbin.org', 'User-Agent': 'python-requests/2.18.1'},
'origin':
'185.136.79.217', 'url': 'http://httpbin.org/get'}
}
```

Также можно посмотреть статус ответа и можно сравнить этот ответ с набором ответов из библиотеки requests.

```
# # Response Status Codes
print(r.status_code)
print(r.status_code == requests.codes.ok)
```

```
200

True
```

Иногда очень удобно при некорректном HTTP-ответе получать исключение.

```
bad_r = requests.get('http://httpbin.org/status/404')
print(bad_r.status_code)
bad_r.raise_for_status()
```

```
404

---------------------------------------------------------------------------

HTTPError                                 Traceback (most recent call last)
<ipython-input-14-9b812f4c5860> in <module>()
      1 bad\_r = requests.get('http://httpbin.org/status/404')
      2 print(bad\_r.status\_code)
----> 3 bad\_r.raise\_for\_status()

/Users/alexander/Development/deep-completion/env/lib/python3.6/site-packages/
requests/models.py in raise\_for\_status(self)

    935
    936         if http\_error\_msg:
--> 937             raise HTTPError(http\_error\_msg, response=self)
    938
    939     def close(self):
httpError: 404 Client Error: NOT FOUND for url: http://httpbin.org/status/404
```

Можем получить заголовки HTTP-ответа.

```
# Response Headers
print(r.headers)
```

```
{'Connection': 'keep-alive', 'Server': 'meinheld/0.6.1', 'Date': 'Sun, 03 Dec 2017
08:46:02 GMT',
'Content-Type': 'application/json', 'Access-Control-Allow-Origin': '*',
'Access-Control-Allow-Credentials': 'true', 'X-Powered-By': 'Flask',
'X-Processed-Time':
'0.000767946243286', 'Content-Length': '267', 'Via': '1.1 vegur'}
```

Допустим мы хотим обратиться к github.com. Мы обращались к HTTP, но попали на HTTPS. Рассмотрим, почему так происходит: в history появилась история, что был ответ 301, то есть мы были перенаправлены.

```
# Redirection and History
r = requests.get('http://github.com')
print(r.url)
print(r.status_code)
print(r.history)
```

```
https://github.com/

200

[<Response [301]>]
```

Если нам не нужно такое поведение, нужно запретить перенаправление:

```
r = requests.get('http://github.com', allow_redirects=False)
print(r.status_code)
print(r.history)
```

```
301

[]
```

Поиск с помощью символьных выражений

Синтаксис регулярных выражений

Большинство букв и цифр означают просто сами себя.
Если перед буквой стоит обратный слэш, то как правило, это спецсимвол, который означает какое-то специальное действие. **d** обозначает цифру от 0 до 9, **D** означает, что в этом месте регулярного выражения может быть всё что угодно, кроме цифры.
Точка означает, что на ее месте может быть любой сивол, кроме перевода строки.
+ означает, что идущий перед ним символ повторяется один или более раз. ***** означает, что он повторяется 0 или более раз. **?** означает 0 или один раз.
a.*c = ***a***b*c*d***a***b***c**d (от первой a до последней c, как можно больше символов между), a.*?c = ***a***b***c**d***a***b***c**d (от первой a до первой c, все вхождения a...c)
() обозначают группы. Все, что в них находится, регулярные выражения запоминют и потом это можно использовать в дальнейшем. Также группы могут быть вложены.

Примеры

- abc = abc;
- a\db = a0b, a1b, ..., a9b, ba1bc, но не подходят a11b, adb;
- a= azb, aab, a_B, A B, но не подходят a1b, a11b, ab, aaab;
- a.b = a0b, aab, a b, но не подходят ab, a11b, abc;
- a\d?b = ab, a5b, но не подходят a55b, acb, a?b;
- a.*b = ab, a123XYZb, a-b=b;
- a.*?b = ab, a123XYZb, a-b=b.

```
import re
html = "Курс евро на сегодня 68,7514, курс евро на завтра 67,8901"
match = re.search(r'Евро\D+(\d+,\d+)', html, re.IGNORECASE)
rate = match.group(1)
rate
```

Без флага IGNORECASE ничего работать не будет, так как в строке "евро" написано со строчной буквы, а в выражении – с заглавной.
Заменим \D+ на .*, и в нашей строке нашлась последняя часть курса, причём не полностью: от 67 у нас осталась только цифра 7. Мы ищем хотя бы одну цифру, поэтому нам достаточно такого числа.

```
re.search(r'Евро.*(\d+,\d+)', html, re.IGNORECASE).group(1)
```

Если мы добавим знак вопроса, то все снова будет хорошо.

```
re.search(r'Евро.*?(\d+,\d+)', html, re.IGNORECASE).group(1)
```

Давайте попробуем теперь найти использовать вместо re.search findall с теми же самыми регулярными выражениями. Мы найдем оба курса.

```
re.findall(r'Евро\D+(\d+,\d+)', html, re.IGNORECASE)
```

Findall вывел все, что у нас находится в группе. Если же групп в выражении не будет, findall выведет всё, что попало под регулярное выражение.

```
re.findall(r'Евро\D+\d+,\d+', html, re.IGNORECASE)
```

Если же групп в регулярном выражение будет несколько, findall вернет список кортежей. В каждом из кортежей будут все значения групп указанных выражений.

```
re.findall(r'Евро\D+(\d+),(\d+)', html, re.IGNORECASE)
```

Если же мы сделаем вложенные группы, также будут возвращены все группы. Группы нумеруется по открывающимся скобкам: первой группой являются внешние круглые скобки, затем идёт вторая группа и третья группа, в таком порядке они возвращаются.

```
re.findall(r'Евро\D+((\d+),(\d+))', html, re.IGNORECASE)
```

На сайте regex101 можно писать и разбирать регулярные выражения в режиме онлайн.

Символьные классы и квантификаторы

Иногда нам необходимо, чтобы в каком-то месте регулярного выражения был один символ из имеющегося у нас набора.

- **символьный класс**, один из множества = [abcd1234] = [a-d1-4]
- \d = [0123456789] = [0-9]; \D = обратный \d символьный класс = [^\D] = [0-9]
- {1,2} = **квантификатор**, предыдущий символ должен повторяться от одного до двух раз
- {2, 2} = предыдущий символ должен повторяться два раза
- {, 2} = предыдущий символ должен повторяться от нуля до двух раз
- {2, } = предыдущий символ должен повторяться два и более (до бесконечности) раз

Теперь попробуем найти все автомобильные номера внутри имеющегося у нас текста. Он начинается с буквы, причём не любой, а одной из 12 имеющих одинаковое написание в русском и латинском алфавите, потом ид три цифры, потом две буквы, всё из того же самого множества, затем две или три цифры регионов. Для этого отлично подойдет символьный класс.

```
import re
text = '''
Автомобиль с номером А123ВС77 подрезал автомобиль
К654НЕ197, спровоцировав ДТП с участием еще двух
иномарок с номерами М542ОР777 и О007ОО77
'''
pattern = r'[АВЕКМНОРСТУХ]\d{3}[АВЕКМНОРСТУХ]{2}\d{2,3}'
matches = re.findall(pattern, text)
matches
```

Обсудим плюсы и минусы использование регулярных выражений.

- Во многих случаях компактно и просто, но бывают и плохочитаемыми;
- Штатный модуль Python, всегда доступен;
- Можно не только искать, но и заменять
- Подходят не для всего;
- Не слишком быстрые.

Сложный поиск и замена

Рассмотрим специальные символьные классы:

- \w = буква, цифра, _;
- \W = [^\] = обратный , не "буквоцифра";
- \s = пробел [\f\n\r\t\v];
- \S = не пробел;
- \b = граница между \w и \W (пустая строка);
- \B = позиция внутри слова;
- ^= начало строки, $ = конец строки.

У нас есть список ников, которые мы хотим проверить. Считаем что в них могут быть только цифры, буквы и знак подчеркивания. Давайте попробуем написать регулярное выражение. Мы используем метод re.compile, который позволяет один раз скомпилировать регулярное выражение и затем несколько раз его использовать. Это удобно использовать, если оно встречается несколько раз в программе или используется внутри цикла, чтобы каждый раз оно не компилировалось заново.

```
import re
nicknames = ['sU3r_h4XX0r', 'alëna', 'ivan ivanovich']
reg = re.compile(r'\w+')
for nick in nicknames:
    print('{} nickname: "{}"'.format(
        'valid' if reg.match(nick) else 'invalid',
        nick
    ))
```

Когд мы запустим код, все ники будут валидны, хотя в алене есть ё, а в иване ивановиче – пробел. Дело в том, что мы написали матч который ищет соответствии с начала строки, но необязательно идет до конца строки. Исправить выражение можно так:

```
import re
nicknames = ['sU3r_h4XX0r', 'alëna', 'ivan ivanovich']
reg = re.compile(r'^\w+$', re.ASCII)

for nick in nicknames:
    print('{} nickname: "{}"'.format(
        'valid' if reg.match(nick) else 'invalid',
        nick
    ))
```

Теперь найдем в строке все слова, начинающиеся с большой буквы и заканчивающиеся на ”-на”.

```
import re
text = (
    'Анна и Лена загорали на берегу океана, '
    'когда к ним подошли Яна и Ильнар'
)

re.findall(r'[А-Я]\w*на', text)
```

В данном случае найдутся Анна, Лена, Яна и Ильна, потому что мы же не говорим о нашем регулярные выражения что это должны быть последние буквы слова. У нас просто после каких-то букв идет ”-на”. Тогда мы можем добавить границу слова в начало и в конец регулярного выражения, это будет означает что мы действительно ищем слово, которое начинается с большой буквой и заканчивается на ”-на”. Давайте попробуем.

```
re.findall(r'\b[А-Я]\w*на\b', text)
```

Если у нас будет Полина, написанная большими буквами, она не найдется: мы ищем слова оканчивающиеся на маленькие буквы. Давайте исправим. Попробуем написать группу, в которую входит как ”-на” маленькое, так и ”-на” большое.

```
import re
text = (
    'Анна и Елена загорали на берегу океана, '
    'когда к ним подошли ЯНА, ПОЛИна и Ильнар'
)
re.findall(r'\b[А-Я]\w*(на|НА)\b', text)
```

Нам вернутся только окончания слов, потому что findall, если находит группу внутри строки, возвращает только то, что попало в группу. Это можно исправить, используя специальный синтаксис ?:, это будет значить, что то, что внутри скобок группируется, но не запоминается.

```
re.findall(r'\b[А-Я]\w*(?:на|НА)\b', text)
```

Beautiful Soup

Введение в Beautiful Soup

Beautiful Soup – это модуль для извлечения данных из HTML и XML, в том числе из документов с ”плохой” разметкой (незакрытые теги, неправильные атрибуты и так далее).

LXML строит дерево синтаксического разбора, по которому можно искать и манипулировать различными данными из HTML и XML. HTML/XML – это дерево; набор тегов, которые могут быть вложены друг в друга, у каждого могут родители, дети и так далее.
Для работы Beautiful Soup нужен парсер – это такой модуль, который умеет непосредственно с HTML/XML, разбирать его, а затем Beautiful Soup преобразует его в дерево. Это может быть, например, html.parser, который прямо встроен в Python. Это неплохой модуль: он имеет среднюю скорость разбора, достаточно лоялен к незакрытым тегам и неправильной разметке, но он не умеет работать с XML.
Поэтому мы будем использовать сторонний модуль lxml. Его надо ставить отдельно. Он быстрый, лоялен к некачественной разметке и при этом он умеет работать с XML.
Таким образом, Beautiful Soup представляет текстовую строку HMTL/XML страницы в виде объекта Python, с которым потом удобно работать, обращаясь к методам модуля soup для получения атрибутов.
Чтобы установить Beautiful Soup, нужно написать **pip install beautifulsoup4 lxml**.

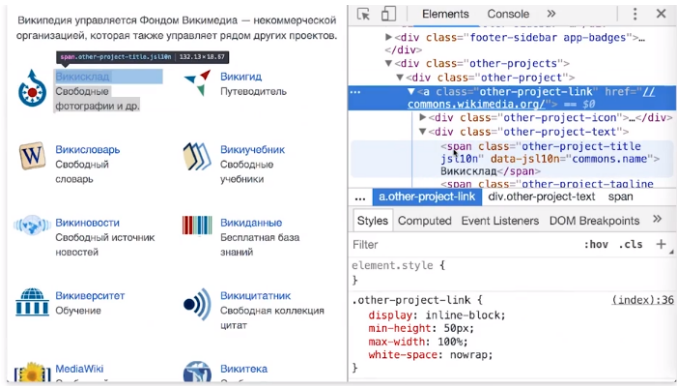
Пусть есть HTML-код, состоящий из тега body и нескольких вложенных в него параграфов со своими атрибутами. Достаточно передать этот XML как строку в объект Beautiful Soup.

```
▼<body>
  ▼<p class="text odd">
      "
        first "
      <b>bold</b>
      " paragraph
      "
    </p>
  ▶<p class="text even">…</p>
  ▶<p class="list odd">…</p>
  </body>
```

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html, 'lxml')
soup.body.p.b.string  # 'bold'
soup.p['class']  # ['text', 'odd']
soup('p')[1]['class']  # ['text', 'even']
```

Теперь попробуем с помощью Beautiful Soup распарсить главную страницу Википедии и из неё

ссылки на другие проекты Википедии.
Посмотрим на HTML-код страницы.



Заметим, что ссылки на другие проекты находятся внутри тега a, у которого установлен класс other-project-link и непосредственно сами ссылки находятся в атрибуте href.
Сделаем парсинг страницы с помощью регулярных выражений. Будем искать открывающийся тег a, внутри которого будем пропускать что угодно, кроме закрытия тега, а затем мы будем искать слово other-project-link. Потом мы будем опять пропускать после него что угодно, кроме закрытия тега, далее будем искать слово href и внутри него мы будем пропускать все, кроме кавычек. В итоге получим список ссылок на все проекты Википедии.

```
import requests
resp = requests.get('https://www.wikipedia.org')
html = resp.text

import re
re_links = re.findall(r'<a[^>]+class="[^"]*other-project-link[^>]+href="([^"]+)',
 ↪  html)
re_links
```

Но получилось довольно сложное выражение, плюс мы читерим:

1) не говорим, что ищем класс other-project-link;

2) считаем, что в href содержимое заключено в двойные кавычки.

Попробуем сделать тоже самое с помощью Beautiful Soup. Нам надо сконструировать Python-объект, передав XML модулю Beautiful Soup и распарсив его с помощью lxml. Теперь в этом объекте soup есть разобранный HTML главной страницы Википедии. Дальше нам нужно воспользоваться методом findall есть, который найдет все теги a с классом other-project-link. Из

тегов нам нужно получить получить содержимое их атрибутов href. Для этого нам достаточно просто перебрать их и заключить результат в генератор списков. Так мы получим такой же результат, как и с помощью регулярных выражений.

```
from bs4 import BeautifulSoup
soup = BeautifulSoup(html, 'lxml')
bs_links = soup('a', 'other-project-link')
bs_hrefs = [link['href'] for link in bs_links]
bs_hrefs
```

При помощи Beautiful Soup все выглядит более красиво и понятно.

Обзор методов модуля Beautiful Soup

Рассмотрим довольно простой HTML, который состоит из тега <body>, у него есть некоторые атрибуты, и вложенных в него трех параграфов, у которых тоже есть атрибут class. В каждый из параграфов тоже вложены различные теги и просто текстовые строки. Мы передаем этот HTML в объект в модуль Beautiful Soup и получаем объект Soup, который и представляет тот самый python-объект для удобной работы с HTML. Выведем его.

```
html = """<!DOCTYPE html>
<html lang="en">
  <head>
    <title>test page</title>
  </head>
  <body class="mybody" id="js-body">
    <p class="text odd">first <b>bold</b> paragraph</p>
    <p class="text even">second <a href="https://mail.ru">link</a></p>
    <p class="list odd">third <a id="paragraph"><b>bold link</b></a></p>
  </body>
</html>
"""
from bs4 import BeautifulSoup
soup = BeautifulSoup(html, 'lxml')
soup
```

Есть также метод модуль method prettify, который выведет его красиво. Он отформатирует его с отступами.
Написав soup.p, мы обратимся к первому тегу <p> внутри HTML. Это BS4.element.tag.
Давайте обратимся к тегу <b> внутри тега <p>. Поскольку внутри тега <b> у нас есть текстовая строка, она уже не является тегом, а просто строка, мы можем написать p.b.string, и тип у нее будет строка, строка Beautiful Soup.

```
soup.p.b.string
type(soup.p.b.string)
```

Это не простая строка, хотя, если мы ее выведем просто, она будет выглядеть как обычная строка bold на содержимое тега. Тем не менее, в отличие от обычной строки, она позволяет использовать методы для перехода к следующему элементу, предыдущему, то есть дополнительные методы, которых нет у простой строки. Если мы удалим отсюда p, то ничего не изменится, потому что Beautiful Soup найдёт первый тег <b>, а это всё тот же самый тег.
Давайте выведем содержимое атрибута class.

```
soup.p['class']
```

Мы получим список классов, которые присвоены этому тегу. У нас в данном случае два класса text и odd. Но даже если бы у нас был один класс, мы получили все равно бы список, да с ним одним, но это был бы список, а не строка. Потому что в спецификации HTML класс – это такой атрибут, который может содержать много значений. Если атрибут может содержать много значений, то Beautiful Soup всегда будет его выводить как список. А, например, атрибут id всегда может содержать только одно значение, и поэтому Beautiful Soup будет всегда возвращать его в виде строки. Даже если мы допишем сюда через пробел еще какие-то значения, по-прежнему будет возвращена одна строка, не будет списка, потому что в спецификации HTML ID не может содержать много элементов.
Давайте еще попробуем, например, вывести родительский тег. Вот мы возьмем, например, тег <b> и выведем его родителя, для этого достаточно написать parent, и это будет наш тег <p>, который является родительским.

```
soup.p.parent.name
```

Также мы можем вывести всех родителей тега <b>. Для этого достаточно написать parents, но это будет генератор. Соответственно, например, если мы захотим вывести все названия тегов, которые идут выше нашего тега <b>, мы можем написать tag.name for tag in parents и сделать из этого список.

```
[i.name for i in soup.p.b.parents]
```

Также мы можем попробовать получить следующий элемент, идущий, например, за нашим тегом <p>.

```
soup.p.next
```

Следующим элементом будет, на самом деле не следующий параграф, а следующий непосредственно за ним. В данном случае first, то есть включая вложенные теги. Если мы напишем p.next.next, то мы получим, очевидно, наш тег <b>.
А если мы хотим все-таки получить следующий элемент, исключая вложенные, то есть, тот, который идет непосредственно за этим <p>, а не внутри него, мы можем написать next_sibling, но результат нас может слегка удивить, потому что непосредственно за этим тегом <p> идет не следующий тег <p>, а перевод строки, потому что у нас всё отформатировано.

```
soup.p.next_sibling
```

Также мы можем получить все вложенные теги внутри какого-то тега. Для этого мы можем написать contents, и нам будет выведен список, состоящий из вложенных как тегов, так и строк, именно в виде списка.

```
soup.p.contents
```

Если мы хотим получить не список, а генератор, то contents можно заменить на children. Мы получаем генератор, соответственно, если мы передадим конструктору списков, то мы получим то же самое, что мы получали с помощью contets.

```
soup.p.children

list(soup.p.children)
```

Сложный поиск и изменение с Beautiful Soup

```
html = """<!DOCTYPE html>
<html lang="en">
  <head>
    <title>test page</title>
  </head>
  <body class="mybody" id="js-body">
    <p class="text odd">first <b>bold</b> paragraph</p>
    <p class="text even">second <a href="https://mail.ru">link</a></p>
    <p class="list odd">third <a id="paragraph"><b>bold link</b></a></p>
  </body>
</html>
"""
from bs4 import BeautifulSoup
soup = BeautifulSoup(html, "lxml")
```

Пусть мы хотим получить родителя, находящегося вверх по иерархии и отфильтровать родителей по какому-то свойству. Мы можем найти тэг body среди родителей элемента b и вывести его id.

```
soup.p.b.find_parent("body")["id"]
```

Есть метод find_next_sibling, который ищет только тэги и позволяет отфильтровать их по какому-то параметру. Можем найти не просто следующего соседа, а только того, у которого есть класс odd.

```
soup.p.find_next_sibling(class_="odd")
```

Также можно использовать метод find_next_siblings. Получим генератор списка, где будут только теги.

```
list(soup.p.next_siblings)
```

Для поиска с фильтрацией внутри какого-то тега можно использовать метод find.

```
soup.p.find('b')
soup.find(id='js-body')['class']
soup.find('b', text='bold')
```

Если мы захотим найти все теги p, нужно использовать метод find_all.

```
soup.find_all('p')
```

Также мы можем писать конкретно, что же мы ищем.

```
soup.find_all('p', 'text odd')
```

Если мы хотим найти тег, где есть odd и text, неважно в каком порядке, мы должны написать p.text.odd.

```
soup.select('p.odd.text')
```

Допустим, мы хотим найти третий из тегов p.

```
soup.select("p:nth-of-type(3)")
```

Если мы хотим найти тег b внутри тега a, то стоит вопользоваться такой структурой.

```
soup.select("a > b")
```

Можем совместить регулярные выражения с Beautiful Soup. Мы хотим найти все теги, которые начинаются с буквы b. Для этого можем импортировать модуль re и написать регулярное выражение.

```
import re
[i.name for i in soup.find_all(name=re.compile('^b'))]
```

Или если мы хотим найти не один тег, а теги из какого-то списка, например все теги a и теги b, мы можем передать просто список.

```
[i for i in soup(['a', 'b'])]
```

Также с помощью Beautiful Soup можно изменять HTML-теги.

```
tag = soup.b
tag

tag.name='i'
tag['id'] = 'myid'
tag.string = 'italic'
soup.p
```

Давайте попробуем распарсить главную страницу новостей mail.ru. Здесь есть какие-то секции, состоящие из заголовка и внутренних новостей на эту тему. Давайте мы попробуем собрать такой список, в котором была бы секция и все новости, которые в неё входят.
Секция – это span с классом hdr_inner. К каждой секции пристыковать все ссылки на материал, который находится внутри секции. Сделаем кортеж: первым элементом будет секция, вторым – набор ссылок.

```
import requests
result = requests.get("https://news.mail.ru/")
soup = BeautifulSoup(result.content, "lxml")
soup

[
    (
        section.string,
        [
             header.string for header in section.find_parents()[4].find_all(
                 class_=['newsitem__title-inner','link__text', 'collections__title',
                 'photo__title']
             )
        ]
    ) for section in soup(class_="hdr__inner")
]
```

Работа через API

Работа через Web-API

API – программный интерфейс приложения. Это – какой-то интерфейс для взаимодействия одних программ с другими.

Когда мы говорим об API в применении к вебу, это называется Web-API. Как правило, это URL или набор URL, на которые мы можем делать HTTP-запросы, передавая свои данные, и получать в ответ какие-то данные, отформатированные, как правило, в JSON или XML. Также есть связанный термин RPC – удаленный вызов процедур. Это когда программа на одном компьютере вызывает методы или функции, расположенные на другом компьютере. При этом данные передаются по сети, а ответ также передается по сети назад.
То, что используются сети, и то, что используются программы, запущенные на другом компьютере, может скрываться от конечного разработчика.
В применении к вебу, как правило, одним из вариантов RPC может являться SOAP – Simple Object Access Protocol. Это текстовый протокол, внутри него лежит XML, а в качестве транспорта используются HTTP.

Также в этом смысле может использоваться rest. Это такой архитектурный стиль, когда есть клиент, сервер, и клиент с сервером обмениваются данными. Также там подразумевается отсутствие состояния, возможность кэшировать и так далее.
Так почему же нужно использовать API? Дело в том, что когда мы парсили HTML, мы, в общем-то, делали неправильно. HTML предназначен для браузера. Там очень много разметки, которая используется чисто для визуального форматирования документа, то есть много мусора. К тому же, дизайнер может захотеть завтра перекрасить какую-нибудь кнопочку и разместить ее в другом месте. Это может изменить весь HTML. И ваш парсер, который вчера работал, завтра уже не будет работать на этом сайте. В то время как API создан специально для работы с программами, там нет лишнего. Там, как правило, находятся чисто данные в XML или в JSON. К тому же API, как правило, не меняется внезапно. API, как правило, подразумевает какой-то удобный формат ответа, XML или JSON, без лишних мусорных данных. И с помощью API, если работа с сайтом это подразумевает, как правило, можно не только читать, но и изменять данные.
У каких же все-таки сайтов есть API? На самом деле, практически у всех информационных сайтов в том или ином виде API присутствует. Например на сайте Центробанка, с которого мы получали курс Евро, есть API, которая возвращает в XML курсы всех валют. И парсить данные из XML гораздо проще и удобнее, чем из HTML, к тому же с Beautiful Soup это будет очень просто. У википедии, которую мы тоже пробовали парсить, есть свое API, оно позволяет искать статьи, изменять их, в общем, в принципе, делать все, что угодно на сайте википедии. У новостей mail.ru API как такового нет, но есть rss-лента, которая, в общем то, тоже является вариантом API.

Практика работы с API

Ранее мы получали курс евро с сайта Центробанка, просто парся его главную страницу. На самом деле, у Центробанка есть API и у него есть документация. Напрмер, есть скрипт для получения курсов за определенный день, куда можно передать какой-то день и получить курсы за него. Давайте попробуем получить данные. Смотрите, мы берем вот этот URL скрипта из документации по API без указания даты и используем его для запроса. Передаем в BeautifulSoup и посмотрим, что мы получаем. Получаем, собственно говоря, XML тот же самый, который мы видели на странице.

```
import requests
from bs4 import BeautifulSoup

resp = requests.get("http://www.cbr.ru/scripts/XML_daily.asp")
soup = BeautifulSoup(resp.content, "xml")
soup
```

Есть блоки валюта, внутри них есть тег CharCode, который содержит название валюты, и его сиблинг, тег валюта, которая содержит value, курс этой валюты.
Мы можем найти блок CharCode с текстом eur для евро и найти ближайший его сиблинг, сиблинг value, и взять его значение строковое. Таким образом мы получим курс.

```
soup.find('CharCode', text='EUR').find_next_sibling('Value').string
```

Также если мы знаем, у каждой валюты есть ID. Вот он. Если мы знаем айдишник, то мы можем сразу найти этот ID и внутри него найти value и получить курс. Мы ищем ID евро. Находим внутри тега валюты тег value и получаем его string.

```
soup.find(ID="R01239").Value.string
```

Дальше давайте познакомимся с API для получения погоды. Есть сайт OpenWeatherMap. Для того, чтобы работать с его API, есть документация по API. У него различные методы, которые позволяют получить текущую погоду, предсказание на несколько дней. Есть подробное описание, как получить погоду в каком-то городе. Есть определенный url, в который надо передать имя города или имя города плюс имя страны. Чтобы работать с API, нужно иметь ключ, для этого нужно зарегистрироваться на сайте. И зарегистрировавшись, перейти в секцию API keys и получить тот самый ключ, который мы будем использовать для работы с сайтом. Из документации мы узнаём url, на которой нам надо обращаться для получения данных по городу, и также какие параметры ему надо передать. Вот мы берем этот url, добавляем его в requests.get, также передаем параметры. Передаем вот этот самый наш APPID, который мы получаем API key внутри секции members. И передаем некоторые режимы, они тоже указаны в документации, хотим получить ответ в exml, указываем mode exml. И единицы измерения — метрические, потому что по умолчанию оно возвращает в имперских. Передаем exml в BeautifulSoup. И можем сначала вывести, какой XML мы получаем. Вот мы получаем через API XML с данными по Москве.

```
resp = requests.get(
    "http://api.openweathermap.org/data/2.5/weather",
    params={
        "q": "Moscow",
        "APPID": "7543b0d800ce423bab3b2f6ad38df30b",
        'mode': 'xml', 'units': 'metric'
    }
)
soup = BeautifulSoup(resp.content, "xml")
soup.temperature['value']
```

Соответственно, если мы посмотрим внимательно эту XML, мы можем заметить, что там есть тэг температуры, внутри у него есть атрибут value, который позволяет получить конкретные значения температуры. Если мы выведем его, то мы получим, что текущая температура 3,74 градуса.
Также используя это API, мы можем получить данные не в XML, а в JSON. Для этого мы оставляем всё как и прежде, только меняем режим с XML на JSON.

```
import requests
resp = requests.get(
    "http://api.openweathermap.org/data/2.5/weather",
    params={
        "q": "Moscow",
        "APPID": "7543b0d800ce423bab3b2f6ad38df30b",
        'mode': 'json', 'units': 'metric'
    }
)
data = resp.json()
data['main']['temp']
```

Кстати, интересно — в XML параметры называются по одному, а в JSON он передает немножко другие названия полей, другие названия параметров. И получить данные из JSON вот мы можем найти секцию main, внутри нее есть секция temp, и там то же самое значение. И соответственно мы обращаемся к словарю, который получается из JSON, и получаем ту же самую температуру.
Можем обращаться к API «ВКонтакте». У «ВКонтакте» огромное API, отличная документация, есть какое-то колоссальное количество методов этих API. Некоторые требует авторизацию, некоторые не требуют.
Посмотрим метод users.get. Он не требует авторизации и позволяет получить данные какого-то пользователя по его какому-то идентификатору, по айдишнику пользователя. Давайте посмотрим, что у нас по этому поводу говорит API. В описании указано, что достаточно обратиться на URL API vk.com/method и указать метод, который вот описан как раз в API. В данном случае мы смотрим метод users.get. Любой метод можно также приписать в конце URL API vk.com/method и выполнить этот тот самый метод. Передаем айдишник пользователя и смотрим, ответ получаем в JSON, что тоже очень удобно, здесь авторизация не требуется для этого метода. Мы берем

пользователя с каким-то айдишником. Это Линдси Стирлинг. Мы также можем указывать дополнительные параметры, по умолчанию он передает только имя или фамилию. Но если мы укажем дополнительные поля, то мы можем получить гораздо больше информации.

```
import requests
resp = requests.get(
    'https://api.vk.com/method/users.get',
    params={ 'user_id': '210700286',
             #'v': '5.68',
             #'fields': 'photo_id, verified, sex, bdate, city, country, home_town,
             has_photo, photo_50, photo_100, photo_200_orig, photo_200,
             photo_400_orig, photo_max, photo_max_orig, online, domain, has_mobile,
             contacts, site, education, universities, schools, status, last_seen,
             followers_count, occupation, nickname, relatives, relation, personal,
             connections, exports, wall_comments, activities, interests, music,
             movies, tv, books, games, about, quotes, can_post, can_see_all_posts,
             can_see_audio, can_write_private_message, can_send_friend_request,
             is_favorite, is_hidden_from_feed, timezone, screen_name, maiden_name,
             crop_photo, is_friend, friend_status, career, military, blacklisted,
             blacklisted_by_me',
           }
)
resp.json()
```

Некоторые методы требуют авторизации, тогда все становится несколько сложнее. Но есть способы, упрощающие работу с API в «ВКонтакте». Вообще для любого крупного сайта на самом деле деле не обязательно напрямую работать с его API через request. Как правило, есть уже готовая библиотека, которая внутри себя работает с его API через request, получает данные. Но оборачивает это в более удобные для нас API. Так, для «ВКонтакте» есть Python модуль VK. Его достаточно поставить pip install vk, и мы можем использовать его для работы с «ВКонтакте». И точно так же API users.get получаю users ID 1.

```
import vk
session = vk.Session()
api = vk.API(session)
api.users.get(user_ids=1)
```

Но вот есть другие методы, которые уже требует авторизации, и с помощью этого модуля сделать их проще. Мы запускаем сессию, передаем идентификационные данные. Правда, для того чтобы работать с этим, нам уже надо зарегистрировать приложение «ВКонтакте», но это всё написано в документации по «ВКонтакте». Передать логин и пароль и получить авторизованную сессию. С помощью авторизованной сессии мы, например, можем получить все группы пользователя.

```
session = vk.AuthSession(
    app_id=6238794,
    user_login='**********',
    user_password='**********'
)
api = vk.API(session)
api.groups.get(extended=1)
```