Работа с Интернет в Python

д-р Филип Андонов

7 юли 2022 г.

Съдържание

- библиотека urllib
- requests
- Сокети
- Изпращане на електронни писма

Библиотека urllib

Тази библиотека предоставя възможности за достъп до файлове през мрежа по начин, еквивалентен с достъпа до локални файлове. С едно извикване всичко, което може да бъде идентифицирано с URL може да бъде получено в програмата на Python. С помощта на urllib можем да изтегляме уеб страници, да извличаме информация, да автоматизираме търсения и много други.

Библиотека urllib

Отварянето на отдалечен файл е еквивалентно на отварянето на локален файл. Разликата е, че файлът може да се отвори само за четене. За да отворим отдалечен файл, използваме функцията urlopen от модула urllib.request. Върнатият файлоподобен обект поддържа методите close, read, readline, readlines, geturl() и info(). На ред 2 викаме метода readlines, връщаш ни списък от редовете на съдържанието. На ред 4 използваме decode, защото върнатият тип е bytes, а ние искаме да работим с него като със str.

Listing 1: Изтегляне на уеб ресурс

```
1  from urllib.request import urlopen
2  webpage = urlopen('http://weather.sinoptik.bg/
       sofia-bulgaria-100727011?location').readlines
       ()
3  for l in webpage:
4      line = l.decode()
5      if '<meta property="og:description" content='
           in line:
6          lft = line.find('Current weather')
7          rght = line.find('Detailed')
8          weather = line[lft:rght]
9          print(weather)
10         break
```

Ако искаме urllib да изтегли и съхрани отдалечен файл вместо вас, можем да използваме urlretrieve. Тази функция връща кортеж (filename, headers), където filename е името на локалния файл (създава се автоматично от urllib), а headers съдържа информация за отдалечения файл. За да изберем името за локалното копие, можем да го подадем като втори параметър. Трябва да имаме предвид, че тя се води „legacy“ и може да бъде премахната. За съжаление, към момента няма заместител, който да прави същото нещо с толкова малко код.

Listing 2: Изтегляне на файл от уеб

```
1 from urllib.request import urlretrieve
2 webpage = urlretrieve('http://upload.wikimedia.
    org/wikipedia/commons/9/93/1_FW_F-22
    _Raptor_participates_in_Red_Flag.jpg','F-22.
    jpg')
```

Помощни функции

За да можем да използваме тези функции, е необходимо да внесем urllib.parse.

quote(string[, safe])
Връща низ, в който всички специални символи са заменени с URL-дружелюбни версии (например ~ става %7E). Това е полезно, когато имаме символен низ, който може да съдържа специални символи и искаме да го използваме като URL. Параметърът safe съдържа символите, които не трябва да се кодират – по подразбиране е '/'.

quote_plus(string[, safe])
Работи по същия начин като quote, но заменя интервалите със знак плюс.

unquote(string)
Обратното на quote.

unquote_plus(string)
Обратното на quote_plus.

urlencode(query[, doseq])
Конвертира съответствие или кортеж от два елемента в URL кодиран низ, който може да се ползва за CGI заявки.

Requests

Requests е модул, който не е част от стандартната библиотека. Той обаче допълнително улеснява някои от функционалностите, предоставени от urllib. Инсталацията му е лесна – ако имаме pip пишем просто

```
pip install requests
```

Името на модула, с който трябва да го внесем в нашата програма също е requests.

Методи

Методите на модула съответстват на HTTP методите: get, post, put, delete, head и options. Когато се обърнем към даден URL адрес, например с get, requests връща обект – отговор, който съдържа всичко, пратено от сървъра. За отговори в JSON формат можем да използваме вградения декодер на модула r.json().

Listing 3: Request вместо urllib

```
1  import requests
2
3  def download_page(url):
4      r = requests.get(url)
5      return r.text
6
7  API_key = 'XXXXXXXXXXXXXXXXXXXX'
8  f = open('weather.json','w')
9  answer = download_page('http://api.openweathermap
       .org/data/2.5/weather?q=London,uk&APPID='+
       API_key)
10 f.write(answer)
11 f.close()
```

Атрибути на отговора

Съдържанието на отговора се намира в атрибута text, а заглавните части – в headers. В атрибута enconding се съдържа текстовото кодиране. В този атрибут може да се пише, като записаното кодиране се използва за всички следващи извиквания на r.text. Ако отговорът не е текст, можем да използваме r.content, за да получим достъп до него като редица от байтове.
Силата на модула е лекотата, с която се извършват не-тривиални обръщения – с поставяне на параметри в HTTP POST обръщение и заглавната част на обръщението.

Listing 4: HTTP POST с Requests

```
1 import requests
2 headers = {'user-agent': 'python scraper 0.1'}
3 data = {'key':'value'}
4 r = requests.post('http://httpbin.org/post',
    headers = headers, data = data)
```

Статус кода на отговора може да се достъпи чрез атрибута r.status_code, а заглавната част на отговора - с атрибута headers.

За да изпратим бисквитки на сървъра (изключително полезно), просто трябва да подадем бисквитката като аргумент на обръщението към него.

Listing 5: Изпращане на бисквитки с Requests

```
1 import requests
2 url = 'https://httpbin.org/cookies'
3 cookies = dict(cookie_test = 'successful')
4 r = requests.get(url, cookies=cookies)
5 print(r.text)
```

Когато се получи отговор от сървър, съдържащ бисквитка, тя се съдържа в RequestsCookieJar. Ние обаче можем да си направим собствен и да събираме там бисквитки, които да важат за различни домейни и пътища. Именно такива буркани с бисквитки могат да се подават на обръщение към сървър, както е показано от следния пример.

Listing 6: Кутия с бисквитки с Requests

```
1 import requests
2 url = 'https://facebook.com'
3 r = requests.get(url)
4 jar = requests.cookies.RequestsCookieJar()
5 for cookie in r.cookies:
6     jar.set(cookie.name, cookie.value, domain =
          cookie.domain, path = cookie.path)
7 r = requests.get(url, cookies=jar)
8 print(r.text)
```

Сокети

Сокетите са основен компонент на мрежовото програмиране. Те представляват информационен канал с програма от двата края. Програмите може да са на различни свързани в мрежа компютри и могат да си пращат информация през сокет. Сокетите са два варианта – сървърни и клиентски. След създаването си сървърният сокет чака за връзка. Той слуша на конкретен мрежов адрес (комбинация от IP адрес и порт), докато не се свърже клиентски сокет с него. Сървърните сокети са по-трудни, защото трябва да чакат за връзка от клиентски сокет и да обработват множество връзки.

Сокетът в Python е екземпляр на класа от едноименния модул. Получава максимум три параметъра при създаване:

- адресната група (по подразбиране socket.AF_INET)
- дали е поток (по подразбиране socket.SOCK_STREAM) или дейтаграмен сокет (socket.SOCK_DGRAM)
- протокол (по подразбиране 0).

За създаването на стандартен сокет нямаме нужда да подаваме никакви параметри. Сървърният сокет използва методите си bind и call, за да слуша на даден адрес. Клиентският сокет може да се свърже към сървъра със своя метод connect със същия адрес, който е използван и от bind. Адресът е кортеж във формата (host, port).

Методът listen приема един аргумент, който е дължината на бек-лога (броя връзки, позволени да се трупат на опашката, очаквайки приемане, преди новите връзки да започнат да се отхвърлят). След като сървърният сокет слуша, той може да приема клиентски връзки. Това става с метода accept. Този метод ще е блокиран (ще чака), докато не се свърже клиент и след това ще върне кортеж от вида (client, address), където client е клиентския сокет и address е адрес. Сървърът прави с клиента каквото се очаква и започва да чака за нови връзки. Това обикновено се прави в безкраен цикъл.

Слайд

Сокетите имат два метода за приемане и изпращане:

- send(string_to_send)
- recv(max_bytes_to_recieve)

Ако не сме сигурни каква стойност да използваме, 1024 е добра идея. Трябва да имаме предвид обаче, че номерата на някои портове не са налични за употреба. В Linux и Unix ни трябват root права за да използваме портове под 1024. Тези портове се използват от стандартни услуги като порт 80 за уеб-сървър. Също така, ако спрем сървъра с Ctrl-C, ще трябва да почакаме малко, за да можем да ползаме същия порт отново. За да илюстрираме използването на сокети, ще създадем сървър, който получава задачи под формата на низ и ги добавя към списък със задачи, който също така връща на клиента, свързан към него. Клиентът от своя страна се свързва към сървъра и изпраща низ, след което слуша за отговор – списък със задачи.

В Python 3 не се разрешава неявно конвертиране от низ в байтов поток, затова конкатенацията изглежда по този начин.

Listing 7: Прост сокет сървър

```
1 import socket
2 s = socket.socket()
3 host='127.0.0.1'
4 port = 1234
5 s.bind((host, port))
6 s.listen(5)
7 tasks = []
```

```
8  while True:
9      c, addr = s.accept()
10     print(addr, "connected")
11     tasks.append(c.recv(1024))
12     print(tasks)
13     for task in tasks:
14         c.send(task + str.encode(", "))
15     c.close()
```

Трябва да имаме предвид, че не е задължително всичко, което е пратено да бъде получено по същия начин. Сървърът може да прати един низ, но той да се разкъса на два или повече пакета. Поради това четенето трябва да се прави с буфер, в който полученото съдържание се добавя, докато не разберем, че сме стигнали до края. Изпращаме данните след като използваме str.encode да ги превърнем от str в byte. Променливата full_reply е от тип bytearray, което е изменим вариант на bytes.

Listing 8: Прост сокет клиент

```
1 import socket
2 s = socket.socket()
3 host = '127.0.0.1'
4 port = 1234
5 s.connect((host, port))
6 task = input("enter a task: ")
7 s.send(str.encode(task))
8 print("=====tasks=====")
```

```
full_reply=bytearray("", "UTF-8")
reply = s.recv(10)
full_reply += reply
while reply:
    reply = s.recv(10)
    full_reply += reply
print(full_reply)
```

Сокетите са полезни, но Python отива една стъпка по-напред и предоставя SocketServer - модул с класове, които позволяват писането на сложни сокет-базирани сървъри с много малко код. Този модул съдържа четири базови класа: TCPServer, UDPServer, UnixStreamServer и UnixDatagramServer. Най-важният (и ползван) е TCPServer. Повечето работа по създаването на SocketServer е дефинирането на клас за обработка на заявките. Това е производен клас на класа BaseRequestHandler от модула SocketServer. Целта на всеки обработчик на заявки е да обработва единична клиентска заявка за времето на връзката между клиента и сървъра. Това се реализира в метода handle на обработчика.

Методите на BaseRequestHandler клас имат достъп до следните три члена:

- request: Сокет обект, представляващ клиентска заявка – това е същият обект, който се получава от socket.accept
- client_address: Кортеж, съдържащ име на хост и порт, към който ще се изпращат сървърните данни.
- server: Референция към SocketServer-а, създал обекта за обработка на заявки

Ако работим с потоци (а TCPServer ползва именно потоци), то можем да ползваме класа StreamRequestHandler. Инстанциирайки StreamRequestHandler вместо BaseRequestHandler дава достъп до файло-подобни обекти, които позволяват четенето и писането в сокет връзки. Те съответно са:

- rfile: Файлът, в който постъпват данните от сокета. Съответства на това, което получаваме от request.makefile('rb').
- wfile: Файлът, данните в който се изпращат през сокета. Съответства на това, което получаваме от request.makefile('wb').

Listing 9: socketserver клас

```
1  import socketserver
2  class RequestHandler(socketserver.
       StreamRequestHandler):
3    def handle(self):
4      l = True
5      while l:
6        l = self.rfile.readline().strip()
7        if l:
8          self.wfile.write(l[::-1] + '\n')
9  hostname = '127.0.0.1'
10 port = 3456
11 server = socketserver.ThreadingTCPServer((
       hostname, port), RequestHandler)
12 server.serve_forever()
```

Множество връзки

Всичко разгледано дотук касае синхронни връзки – в даден момент само един клиент може да се свърже и да му бъде обработена заявката. Ако заявката отнема време (например е чат сесия), е важно да може да се обработват няколко връзки едновременно. Модулът SocketServer дефинира два класа за обработка на множество връзки едновременно: ThreadingMixIn и ForkingMixIn. Клас SocketServer, наследяващ ThreadingMixIn ще създава автоматично нова нишка когато получи заявка. Наследник на ForkingMixIn автоматично ще fork-ва нов суб-процес да обработва всяка постъпваща заявка. ThreadingMixIn е по-ефективен и по-преносим от употребата на суб-процеси.

Изпращане на електронни писма

Функционалността за работа със SMTP се намира в модула smtplib а SMTP клиентски сесиен обект, който да се използва за изпращане на писма до свързани машини с SMTP или ESMTP демони. Използването е различно сложно в зависимост от това какво искаме да направим. В примера по-долу първо създаваме SMTP обект, който осъществява връзка със сървър. Ако искаме връзки към няколко сървъра, е необходимо да създадем по един обект за всеки сървър. След това използваме метода login, за да влезнем в сървъра. За да изпратим прост текст, остава само да извикаме метода sendmail с аргументи адрес на подателя, адрес на получателя и низ, съдържащ самото съобщение.

Listing 10: Изпращане на електронно писмо

```
1 import smtplib
2 server = smtplib.SMTP('smtp.gmail.com', 587)
3 server.login("youremailusername", "password")
4 msg = "Hello!"
5 server.sendmail("you@gmail.com", "target@example.
    com", msg)
```

В този пример обаче пропуснахме няколко неща. Не дефинирахме заглавието на писмото и нямахме възможност да прикачим файл. Затова в следващия пример ще реализираме изпращане на писма с прикачени файлове през Gmail. На 15-и ред създаваме контейнера на писмото. На ред 25 използваме MIMEApplication, който е клас, производен на MIMENonMultipart и се използва за представяне на обекти - MIME съобщения от типа application.

Listing 11: Изпращане на писмо с всички атрибути

```
1  import smtplib
2  import string
3  from os.path import basename
4  from email.mime.image import MIMEImage
5  from email.mime.multipart import MIMEMultipart
6  from email.mime.text import MIMEText
7  class gmailSender:
8    def __init__(self, user, passw):
9      self.s = smtplib.SMTP('smtp.gmail.com',587)
10     self.s.ehlo()
11     self.s.starttls()
12     self.s.ehlo
13     self.s.login(user, passw)
```

```
14    def send_via_gmail(self, subject, to, sender,
          filepath, msgtxt):
15        msg = MIMEMultipart()
16        me = sender
17        recipients = to
18        msg['Subject'] = subject
19        msg['From'] = sender
20        msg['To'] = recipients
21        msg.attach(MIMEText(msgtxt))
22        if filepath:
23            for f in filepath or []:
24                with open(f, 'rb') as fil:
25                part = MIMEApplication(
26                    fil.read(), Name=basename(f))
```

```
27                part['Content-Disposition'] = '
                      attachment; filename="%s"' %
                      basename(f)
28                msg.attach(part)
29        self.s.sendmail(me, recipients, msg.as_string
              ())

31    def close(self):
32        self.s.quit()
```

Задача

Да се създаде графично приложение с помощта на Tkinter, което изтегля rss файла на сайта slashdot.org (файлът се намира на адрес http://slashdot.org/slashdot.rss). Файлът трябва да се обработи така, че да се извлекат данните заглавие(title) и съдържание(decription) на всеки запис(item). Заглавията ще се показват в списък. При избор на елемент от списъка, съдържанието на новината да се показва в текстов компонент.

Задача

За да направим нещо интересно с данни от Интернет, ще изтегляме статии от wikipedia, които са гео-маркирани и се отнасят за обекти в даден град. Основната функция е downloadCity, която получава име на град и координатите на центъра му. Тя използва download_page, за да се обърне към приложния интерфейс на Wikipedia и да получи статиите, които се намират в радиус от 10 километра от центъра. След това изтегля и самата статия за града като текст. Резултатът от заявката към приложния интерфейс връща json низ, който анализираме и с помощта на filter_article проверяваме дали всяка статия се съдържа като текст в статията за града. Тези, за които това е вярно, се добавят в крайния списък.

Благодаря за вниманието!

Въпроси?

References