Some Common Python Libraries for Web

23

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- **LO 1** Use python-whois and the "requests" library
- **LO 2** Know the object of class Response returned by the request() method
- **LO 3** Understand the use of "cookies"
- **LO 4** Use the requests.Session() class
- **LO 5** Know SSL Cert Verification and using OAuth 2.0
- **LO 6** Use the "tweepy" module
- **LO 7** Understand the use of Beautiful Soup Python library
- **LO 8** Knowing the basics of socket programming in Python
- **LO 9** Learn the basics of FTP and use FTP protocol to download data from the Internet
- **LO 10** Learn the basics of smtplib module to send mail with an SMTP listener daemon

23.1 INTRODUCTION

This chapter discusses a number of concepts related to Internet. For example, there are concepts like: UA- User Agent, http requests, SSL certification, Restful API, which are briefly discussed here.

A large number of open source free libraries are available for doing various activities on the web.

This chapter discusses the following libraries/modules:

1. Python-whois for getting a web page owner information.
2. Libraries "requests" and "BeautifulSoup4"
3. Socket programming

4. ftplib
5. smtplib

This chapter also contains basics of socket programming in Python. Programming at socket level is basically "low- level" programming. It may be noted that a number of Python libraries/modules exist which can do all that socket programming does (and much more in some cases). So the topic of socket programming can be completely skipped without loss of continuity)

23.2 BASICS OF HTTP REQUESTS

This section deals with the basic concepts related to a "http request". It also deals with how to "owner information" of a website using "python-whois" module.

23.2.1 Getting Server/Owner (of web-site) Information Using python-whois

Python has a library called python-whois available for getting information about the server/owner of a website. You can do a pip install as follows:

```
pip install python-whois
```

You may use the whois library to find out details about Google at: https://www.google.co.in/ . The code on Jupyter is as follows:

```
1 import whois
2 print(whois.whois('https://www.google.co.in/'))
```

The output is:

```
1  {
2    "domain_name": "GOOGLE.CO.IN",
3    "registrar": "MarkMonitor Inc.",
4    "updated_date": null,
5    "creation_date": null,
6    "expiration_date": null,
7    "name_servers": [
8      "NS1.GOOGLE.COM",
9      "NS2.GOOGLE.COM",
10     "NS3.GOOGLE.COM",
11     "NS4.GOOGLE.COM"
12   ],
13   "status": [
14     "clientDeleteProhibited",
15     "clientTransferProhibited",
16     "clientUpdateProhibited"
17   ],
18   "emails": null
19 }
```

23.2.2 Python Library "requests"

Python has a requests module to handles http requests. Data exchange in HTTP (Hyper Text Transfer Protocol) has a "Request-Response" Cycle. So a cycle makes a "request" and gets a "response" from a server. Then the first thing to understand is how to make a request. An HTTP request has certain components which are explained in the next section.

23.2.3 Components of a http Request

A request generally has some or all of the following four things:

- **URL (Uniform Resource Locator):** URL stands for Uniform or Universal Resource Locator. URLs are generally web addresses, but they could also refer to other resources like file transfer protocols (FTP) and database access.
- **Headers:** Header provides some additional information about a request.
- **Method:** In general there are four types of requests that a client may make to a server. These are:
  - GET—To request a server to get some resource.
  - POST—To request a server to create some new resource.
  - PUT—To request server to edit some resource.
  - DELETE—To request a server to delete some resource.
- **Body:** The body of a request contains the data that the client wants to send to the server.

The following script uses the "request" module to create a simple request using the "get" method. The get() method of the requests module needs a valid url as a parameter:

```
1  import requests
2  page = requests.get('http://www.gutenberg.org/files/2591/2591-h/2591-h.htm')
3  print(page.text[:300])
4  #Output
5  ï»¿<?xml version="1.0" encoding="utf-8"?>
6
7  <!DOCTYPE html
8     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
9     "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd" >
10
11 <html xmlns="http://www.w3.org/1999/xhtml" lang="en">
12 <head>
13 <title>
14       Fairy Tales, by The Brothers Grimm
15 </title>
```

Line2: Here the get(url) method returns an object of type Response which is pointed to by the variable page.

Line3: The response object, i.e., page has a text property which gives the text returned by the get() method. Using text[:300] limits the text displayed to first 300 characters of the text.

However in real life, whenever you write code for getting data from the net, you must take into account the possibilities of errors, and so must wrap your code in appropriate try-except blocks. Following is a script which includes an additional parameter for the get() function, i.e., timeout=5. The response object returned has a status_code property which will be 200 if all is ok.

```
1  import requests
2  def func(url, timeL= 5):
3      try:
4          r = requests.get(url, timeout =timeL)
5          if r.status_code != 200:
6              print("problem")
7              return None
8          print('response code->', r.status_code)
9          return r
10     except requests.exceptions.RequestException as e:
11         print(e)
12         print("Something went wrong")
13         return None
14 def checkPage(page):
15     if type(page) == requests.models.Response:
16         print('First 100 characters of response page->')
17         print(page.text[:100])
18     else:
19         print('page type is->', type(page))
20 #First call with default time limit 5
21 page1 = func('http://www.gutenberg.org/files/98/98-h/98-h.htm')
22 checkPage(page1)
23 page2 = func('http://www.gutenberg.org/files/98/98-h/98-h.htm', 0.005)
24 #Second call with timeL .005, which is too short so will throw error
25 checkPage(page2)
```

```
26 # OUTPUT
27 response code-> 200
28 First 100 characters of response page->
29 ï»¿<?xml version="1.0" encoding="utf-8"?>
30
31 <!DOCTYPE html
32    PUBLIC "-//W3C//DTD XHTML 1.0 Strict
33 HTTPConnectionPool(host='www.gutenberg.org', port=80): Max retries exceeded with
34 url: /files/98/98-h/98-h.htm (Caused by
35 ConnectTimeoutError(<urllib3.connection.HTTPConnection object at 0x0A50EFB0>,
36 'Connection to www.gutenberg.org timed out. (connect timeout=0.005)'))
37 Something went wrong
38 page type is-> <class 'NoneType'>
```

Lines2-12: This is definition of a function func(url) which takes a url as a parameter. It has a try block (Lines3-9) and an except block(Lines10-12).

Lines5-7: If the status_code is not 200, the function returns a None.200 is the HTTP status code for "OK", i.e., a successful response. (Other common status codes are 402: Forbidden; 404: Not Found, 403: Forbidden, and 500: Internal Server Error.[1])

Lines 8-9: However, if the status_code is 200, the function returns a Response object.

[1]For a list of common responses, see: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2.1

Lines 10-13: If the request does not succeed and throws an error, it will be caught here. Note that the exceptions in requests module are in `requests.exceptions.RequestException`. There are two print statements. The first print, i.e., the string attached to the error object. The second prints the message "Something went wrong".

Lines 14-19: This is a second function which takes as its argument the Response object returned by the requests.get() method. Here also you must check whether you got a valid Response object. The checking is done in line15, where you check the type of the object page and see if it is requests.models.Response. (How I found out the type of the Response object? I printed the type of page using print(type(page) and used the output in line 15)

Lines 21: This is a normal call to func() giving only one parameter, i.e., the URL, so for timeL, the default of 5 is taken. All is OK.

Lines 24: Here on purpose, a very small value of time L, i.e., .005 is given, forcing an error to be raised.

The signatures of the common request methods (7 in all) are as follows[2]:

```
1 get(url,params=None,**kwargs)# Sends a GET request.
2 options(url,**kwargs)# Sends a OPTIONS request.
3 head(url,**kwargs)# Sends a HEAD request.
4 post(url,data=None,json=None,**kwargs)# Sends a POST request.
5 put(url,data=None,**kwargs)# Sends a PUT request.
6 patch(url,data=None,**kwargs)# Sends a PATCH request.
7 delete(url,**kwargs)# Sends a DELETE request.
```

23.2.4 Using requests.request(method, url, **kwargs)

The signature of the requests.request() method is:

```
1 requests.request(method, url, **kwargs)
```

In the above signature, there are three items described in the following list:

- Method: This parameter is the 1st parameter. It could be of any of the seven types, i.e., GET, OPTIONS, HEAD, POST, PUT, PATCH and DELETE.
- URL: This parameter is the URL of the server to which the request is to be made.
- **kwargs: You know that in Python **kwargs stands for "key-value pair". So this stands for the additional key-value pairs that you may want to give to your request object. Some common "keys" for **kwarg of request method are:
  - data: Suppose you want to send some data with your request, then you could add a key-value pair in form data = some_data, where some_data could be a dictionary, file like object, etc.
  - headers: You could send a "dictionary of headers" with this optional key
  - There are many other optional key-value pairs as part of **kwarg that you may send. They are not discussed but can be studied from the online documentation of the requests library.

Some other noteworthy points regarding the requests.request() method are:

- It is a "general method" and therefore can be used for "any of the 7 types" of requests. Note that the requests library also provides all these seven methods as separate methods. So for example you could make a request GET using requests.request('GET', some_url), you could make the same request using requests.get(some_url).

[2]Taken from: https://www.pydoc.io/pypi/requests-2.9.2/autoapi/api/index.html

- The other important thing about the requests.request() method is that it "returns" a "Response" object. This response object also has a large number of attributes and methods through which you can do various things. For example one "attribute" of the Response object is "content" through which you can get the content of the response in bytes.

The complete signature of requests.request(method, url, **kwargs) is as follows:

requests.request(method, url, **kwargs)
Constructs and sends a Request object to the server. The return is a Response object.

1. There are two mandatory parameters: method and url:
   (a) method – method for the new Request object. (It could be of any of the seven types, i.e., GET, OPTIONS, HEAD, POST, PUT, PATCH and DELETE.)
   (b) url – Takes the URL of the server.
2. Parameter **kwargs (All are optional)
   (a) Params: Note "params" is also a parameter. It contains either a Dictionary object or bytes to be sent in the query string as part of the Request.
   (b) Data: The "data" parameter may be Dictionary object, or list of tuples [(key, value)], bytes, or file-like object to send in the body of the Request.
   (c) json: The parameter "json" takes json data to be sent as part of the Request object.
   (d) Headers: The parameter "headers" is a Dictionary object in form of key-value pairs. It consists of HTTP Headers to be sent as part of the Request object.
   (e) Cookies: The parameter "cookies' is either a Dictionary object or a CookieJar object to be sent as part of the Request object.
   (f) Files: The parameter "files" is meant for "uploading files" through the Request object. There are many ways of specifying this parameter. One possible way of uploading a file through the "files" parameter is to give it a Dictionary object of type {file_name: file_like_object}. There are other possibilities also, but they are not discussed here.
   (g) Auth: The parameter "auth" takes a tuple of type (user_name, pass_word) to "authenticate a user. (***Note:*** An example on how to use the "auth" parameter is given at the end of this chapter in "Beyond Text Book" section.
   (h) Timeout: The parameter "timeout" can take a float value indicating the "timeout" for the request.
   (i) allow_redirects: The parameter "allow_redirects" is a bool. If it is set to True, then, the server may direct your request to another url. By default it is set to True. (***Note:*** If you want to know whether your Request was "redirected" by the server, there is a way of doing this. The Response object returned by the requests.request() method has an attribute named "history". By checking the value of this parameter, you can know whether your request was redirected or not
   (j) proxies: The parameter "proxies" takes a Dictionary object. (For details see online documentation)
   (k) verify: The parameter "verify" is for use of SSL certificate of the server being accessed by the client. If set to False, it will not verify the SSL certificate. if True, the SSL cert will be verified. By default it is set to True.

(l) stream: The parameter "stream' is a flag. If it is set to True, then data will not be downloaded in "one go', but rather it will be "streamed", i.e., it will be downloaded in chunks. But if you set it to False, the entire data is downloaded in one go. In general, stream should be set to True.

(m) Cert: The parameter "cert" is for "client side" certificate. Note that the certificate consists of two entities, i.e., the "private key" and the "certificate". You can give a "path to the file" containing the key and certificate in the form of a string, i.e., str. Alternatively you can also give a tuple of two strings containing the paths to the "private key" and the "certificate".

The important thing is to understand how this works. The first two parameters, i.e., method and url are mandatory and must be provided. However, all the rest are optional and have to be provided in keyword= argument form. For example, suppose you want to set a time limit of say 3 seconds for the request to execute, then apart from the first two parameters, you must also provide timeout = 3.

The following script executes the following tasks:

- It takes the url of a text file at gutenberg.org.
- It uses the method "GET".
- It gets a Response object here called resp_obj.
- It prints the various "attributes" of this Response object.
- Of note is an attribute of Response object namely "text". This attribute is of type "string", i.e., "str". This attribute contains the contents of the response in Unicode.

The script is as follows:

```
import requests
# url of a text file at gutenberg taken
guten_url = 'http://www.gutenberg.org/files/98/98-h/98-h.htm'
resp_obj = requests.request(method = 'GET', url = guten_url)
# The Response object here is resp_obj
# It has a number of attributes
print('Encoding->', resp_obj.encoding)
print('status_code->', resp_obj.status_code)
print('url->', resp_obj.url)
print('Cookies->', resp_obj.cookies)
print('Time elapsed->', resp_obj.elapsed)
print('headers->', resp_obj.headers)
print('Links->', resp_obj.links)
# Response object also has a text attribute of type string
print('Response object has attribute text of type->',type(resp_obj.text))
print('Text->', resp_obj.text[200:400])
```

```
# OUTPUT. . .
Encoding-> ISO-8859-1
status_code-> 200
url-> http://www.gutenberg.org/files/98/98-h/98-h.htm
Cookies-> <RequestsCookieJar[]>
Time elapsed-> 0:00:00.249558
headers-> {'Server': 'Apache', 'X-Rate-Limiter': 'zipcat2.php', 'Last-Modified':
```

```
'Mon, 19 March 2018 15:36:46 GMT', 'ETag': '"c0ad56d6"', 'Content-Encoding':
'gzip', 'X-Zipcat': '319060 / 980546 = 0.325', 'Accept-Ranges': 'none', 'X-Frame-
Options': 'sameorigin', 'X-Connection': 'Close', 'Content-Type': 'text/html', 'X-
Powered-By': '3', 'Date': 'Tue, 16 Apr 2019 02:21:59 GMT', 'X-Varnish':
'474783606', 'Age': '0', 'Via': '1.1 varnish', 'Content-Length': '319060',
'Connection': 'keep-alive', 'Vary': 'negotiate, accept-encoding'}
Links-> {}
Response object has attribute text of type-> <class 'str'>
Text-> 99/xhtml" lang="en">
  <head>
    <title>
      A Tale of Two Cities, by Charles Dickens
    </title>
    <style type="text/css">
    <!--
    body { margin:5%; background:#faebd0; text-align:j
```

Note that .cookies, .elapsed, .headers, .links, .text[] are all properties/methods of the Response object returned by the request() method.

Cookies in Python coding: Cookies can be thought of as a Python dictionary, i.e.,consisting multiple key-value pairs.

23.2.5 Using requests.Session()

Note that instead of using requests.request(method = "GET", url = some_url), you could have also used requests.get(url = some_url).

However, one problem with using requests.get() is that if the same website or server has to be called a number of times, then for each call, a new cookie is generated. This is shown in the following code:

```
import requests
# First get()
req1 = requests.get('https://twitter.com/')
# See what cookies twitter provides
print(req1.cookies)
# session cookie is _twitter_sess
# So get cookie _twitter_sess
cookie1 = req1.cookies['_twitter_sess']
print('cookie1->',cookie1)
# Second get(). Same server but different url
req2 = requests.get('https://twitter.com/i/moments')
# Get cookie _twitter_sess for 2nd get()
cookie2 = req2.cookies['_twitter_sess']
print('cookie2->',cookie2)
# Compare the 2 cookies
print('cookie1 same as cookie2?', cookie1 == cookie2)
```

```
# OUTPUT
<RequestsCookieJar[<Cookie _twitter_sess=. . . .(REST TRUNCATED TO SAVE SPACE)
```

```
cookie1->
BAh7CSIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNo%250ASGFzaHsABjoKQHVz
ZWR7ADoPY3JlYXRlZF9hdGwrCIfazBdnATOMY3NyZl9p%250AZCIlZjE4Y2FjYWIyZTFiYWEyMDFmODAz
MDAyZTY4ZGNhNmE6B2lkIiVmYjg1%250AOTZhZjMxOWY0ZWJkYWU4Y2FiMDI3MTgwMjBhNA%253D%253D
--d5659565bbaccc092ecc4d18e3112ad21d29423f
cookie2->
BAh7CSIKZmxhc2hJQzonQWN0aW9uQ29udHJvbGxlcjo6Rmxhc2g6OkZsYXNo%250ASGFzaHsABjoKQHVz
ZWR7ADoPY3JlYXRlZF9hdGwrCOnnzBdnATOMY3NyZl9p%250AZCIlOWY1NTljOTMyM2YyMTA5NjkzYmJh
ZmQ0NWFhNjhmMDM6B2lkIiUyODEy%250AYzFhZTNkOGMwYTNmNTU3NTIxNGQyY2RmYmI0Yw%253D%253D
--0359d6747c96c409901290e5ed59163bd005b8f2
cookie1 same as cookie2? False
```

If you want to have multiple interactions with the same server/website, then it is better to go for a Session() object. (Note it is Session() not Sessions() and it begins with a capital S).

The following code shows how a Session() object is created and also shows that repeated calls with the Session() object keep the cookies intact.

```
import requests
# Create Session() object
session_obj = requests.Session()
# Make 1st get() request
req1 = session_obj.get('https://twitter.com/')
# See what cookies twitter uses.
print(req1.cookies)
# Print shows there is a cookie _twitter_sess
# So get value for cookie '_twitter_sess'
cookie1 = req1.cookies['_twitter_sess']
print('cookie1->', cookie1)
# Make 2nd get() request on same server but different url
req2 = session_obj.get('https://twitter.com/i/moments')
cookie2 = req2.cookies['_twitter_sess']
print('cookie2->', cookie2)
# Compare the 2 cookies
print('cookie1 same as cookie2?', cookie1 == cookie2)
```

```
# OUTPUT
<RequestsCookieJar[<Cookie _twitter_sess= . . . . (REST TRUNCATED TO SAVE SPACE)
cookie1-> BAh7CSIKZmxhc2hJQz. . . . (REST TRUNCATED TO SAVE SPACE)
cookie2-> BAh7CSIKZmxhc2hJQz. . . . (REST TRUNCATED TO SAVE SPACE)
cookie1 same as cookie2? True
```

23.2.6 A brief note on SSL Cert verification

The Requests library uses SSL certificates for SSL verification. This behaviour of Requests is similar to a web browser. The requests module has an attribute certs which gives location of the certs.py file which is further used to locate the certificates. The code is as follows:

```
import requests
print('requests.certs->', requests.certs)
```

```
# OUTPUT
requests.certs-><module 'requests.certs'from'
C:\\ProgramData\\Anaconda3\\lib\\site-packages\\requests\\certs.py'>
```

If you really want to know what is happening under the hood, then have a look at the source code of certs.py file. (You can look it up on your computer or also on github[3].)

23.3 OAuth and tweepy Module

This chapter discusses an authentication protocol called OAuth 2.0. It then discusses the use of this "authentication" protocol through a popular module called tweepy which is used for accessing social mediasite Twitter.

23.3.1 A Brief Note on OAuth 2.0

(a) The need:

Suppose you develop an app (let's call it A) which allows a Twitter user (let's call him U) to collect his statistics or perform some other functions on Twitter, i.e., a service provider (Lets call him S). The old way of doing this would be that the user U of the app A gives his password to the app A, then the app would do some task on the service provider S on behalf of the user. This scheme has a big problem because, then the app A would have access to entire user U's account and be able to do many things on behalf of the user and even change his password on the service provider S. This is where a service like OAuth2 comes in. The scheme works like this: the user U provides his username not to the app A but to the service provider U (Twitter in this example) and the Twitter service provider S then allows the app A to do something on behalf of the user U.

(b) The process:

The OAuth 2.0 service requires that you as an app developer first register with the service provider on the website of the service provider. Once you register your app with the service provider, you will get an API id (Or client id) and also an API secret (also called client secret).

Now that you are registered with the service provider, then when a user uses your app then you as an app present him with a link to the service provider (i.e.,Twitter in this example). The user then provides his username and password not to your app but to the service provider Twitter. Now the app can do some action on behalf of the user without knowing his password.

23.3.2 Module tweepy

In the previous sections requests and then OAuth2.0 were discussed. Python has many modules which can make the process of creating an app easier. One such module is tweepy which has been especially created to interact with popular social media site Twitter.

You can download and install the module by pip install tweepy.

However, before you can use tweepy, you need to create a new app on Twitter. To do this you must first have your own account on Twitter. Thereafter you need to sign in into https://apps.twitter.com/ using

[3]For seeing the certs.py file on github, see: https://github.com/requests/requests/blob/master/requests/certs.py

your Twitter credentials. While creating your app, you may need to give a name_for_app, description, a callback_url, etc. and then you can get a consumer_key or API_key, consumer_secret or API_secret. You can also get an access_token and an access_key.

23.3.3 Using the tweepy Module[4]

The tweepy module has a class OAuthHandler() in a file auth.py[5]. It is helpful to look at the first few lines of the source code of OAuthHandler class. These first few lines are as follows:

```
class OAuthHandler(AuthHandler):
    """OAuth authentication handler"""
    OAUTH_HOST = 'api.twitter.com'
    OAUTH_ROOT = '/oauth/'

    def __init__(self, consumer_key, consumer_secret, callback=None):
#....... REST OF CODE NOT SHOWN ..............................
```

If you look at the above __init__() method, you can see that it takes two parameters `consumer_key, consumer_secret, .`

Further this class has a set_access_token() method. The relevant portion of code is as follows:

```
    def set_access_token(self, key, secret):
        self.access_token = key
        self.access_token_secret = secret
#....... REST OF CODE NOT SHOWN ..............................
```

Note:

The advantage of looking at the source code:
Many of the modules are often not properly documented. So looking at the source code helps to better understand the module. Also in most documentations, you will find the values given for the arguments, but not the actual names of the arguments. Once you see the source code, you can know what arguments are to be given and what values are to be given for these arguments. For example, by looking at the source code of the above method, you can see that it takes two arguments named key and secret.

Further this class, i.e., OAuthHandler() also has a get_username() method which returns the username of the Twitter user.

So you may now write a script which uses the tweepy module:

```
import tweepy
# Put values for your API and tokens
CONSUMER_API_KEY = 'wrxxxxxxxxxxxxxxxxxxxxv'
CONSUMER_API_SECRET = 'fxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxMR7LndeOqTw5rEMI'
ACCESS_TOKEN = '1xxxxxx3-zxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxKb'
ACCESS_TOKEN_SECRET = 'YxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxB'
```

[4]Documentation of tweepy is available at: https://media.readthedocs.org/pdf/tweepy/v3.2.0/tweepy.pdf

[5]You can see the auth.py file on github at: https://github.com/tweepy/tweepy/blob/master/tweepy/auth.py

```
7
8  auth = tweepy.OAuthHandler(
9          consumer_key = CONSUMER_API_KEY,
10         consumer_secret = CONSUMER_API_SECRET)
11 auth.set_access_token(
12         key = ACCESS_TOKEN,
13         secret = ACCESS_TOKEN_SECRET)
14
15 print('user-name->', auth.get_username())
```

```
16 # OUTPUT
17 user-name->26041965.ag
```

The above script creates an object of OAuthHandler() class and then adds the various tokens to it. It then gets the username of the user.

23.3.4 RESTful API Class of tweepy Module

There are two classes of APIs provided by Twitter and accordingly the tweepy module has two types of APIs to deal with them. Very briefly these two types of are called the

- RESTful APIs, and
- Streaming APIs.

The RESTful APIs are for getting that data that is already available in the account of a Twitter user while streaming APIs are for getting data from a user account as and when he publishes something on his account. So RESTful APIs work for the past data while the streaming APIs work for the data as it comes.

In this chapter only the RESTful APIs are discussed. One important point to note is that in order to prevent mass scale harvesting of data, the Twitter app limits how far back you can go back in time (approximately 1 week) and at what rate you can download data, etc.

The API class of tweepy module has many methods. For this exercise, two such methods namely API.home_timeline() and API.user_timeline() are used. These two methods will return what is an object of class RestfulSet. This object is actually an extension of the Python list class, so you can think of it as a list. So this object is a sequence just like a list. Further this object (of type RestfulSet) in turn contains other objects which are objects of class Status. (In the tweepy module, a tweet is represented by a Twitter.models.Status object from the python-twitter module.)

Add the following lines of code to the previous example (The Twitter handle for the Python software foundation is @ThePSF, so you may use the user_timeline() method to get tweets from this handle.):

```
1  # This script is continuation of previous. It uses auth object created earlier
2  #Create an object of class API
3  api = tweepy.API(auth_handler = auth)
4  # Confirm that the API object created ie api is indeed of type API
5  print('api->', type(api))
6  # Use API.home_timeline() method to get 10 Status objects in list
7  data_home = api.home_timeline(count = 10)
8  # Confirm that the type of object returned ie data_home is of class ResultSet
9  print('API.home_timeline() method data type->', type(data_home))
10 # There is a python module collections and it has a class Sequence
```

```
# This sequence class can be used to check if an object is a sequence
print('is data_home a sequence?', isinstance(data_home, Sequence))
# Confirm that type of objects contained in returned list are of class Status
print('what does data_home contain?->', type(data_home[0]))
print('Number of Status objects got from my account->', len(data_home))

# Use API.user_timeline()
# ThePSF is screen name for The Python Foundation
# twitter does not permit downloading more than 200 Status objects
data_PSF = api.user_timeline(
                                screen_name = 'ThePSF',
                                count = 200)
print('data_PSF is sequence?', isinstance(data_PSF, Sequence))
print(type(data_PSF[0]))
print('Number of Status objects got from ThePSF->', len(data_PSF))
```

```
# OUTPUT
api-><class'tweepy.api.API'>
API.home_timeline() method data type-><class'tweepy.models.ResultSet'>
is data_home a sequence? True
what does data_home contain?-><class'tweepy.models.Status'>
Number of Status objects got from my account->10
data_PSF is sequence? True
<class'tweepy.models.Status'>
Number of Status objects got from ThePSF->200
```

The above code explains how the tweepy module works.

If you want to go into the details of how the code works, you may have a look at the api.py file available here[6]. Further if you want to see the source code of the RestfulSet class and the Status class looks like, you may see the models.py file available here[7].

23.4 BEAUTIFUL SOUP

Python has a library called Beautiful Soup to extract data from HTML and XML files.

BeautifulSoup4 is the latest version and it can be installed by using the following command on Anaconda prompt:

```
pip install BeautifulSoup4
```

Note that if BeautifulSoup4 is already installed, you will get the message shown in Figure 23.1.

FIGURE 23.1 Screen shot when BeautifulSoup4 is already installed.

[6]See: https://github.com/tweepy/tweepy/blob/master/tweepy/api.py

[7]See: https://github.com/tweepy/tweepy/blob/master/tweepy/models.py

Once you have downloaded these packages, you need to import them into your script just like any other Python module. Do note that if you installed with pip, you'll need to import from bs4. If you download the source, and built it, you will need to import from BeautifulSoup. In the author's case, the import was done from bs4.

Note that Beautiful Soup (Acronym BS) is a Python library for getting data out of markup languages like HTML and XML. HTML as a markup language has a huge number of tags. The complete list of HTML tags can be seen at this footnote[8].

BS uses what is called parsers. The default parser is "html.parser". The others common ones are "lxml" and "html5lib". Default "html.parser" means that if no parser argument is specified then this will be used, but it is a good practice to always specify a parser in the BeautifySoup() constructor. Typical constructor will be BeautifylSoup(markup, "html.parser") or name of other parsers. Note in place of markup you could always specify a web address of a filepath pointing to an html page. Note if the lxml or html5lib parsers are not installed, you may do a `pip install lxml` or `pip install html5lib`.

Following is a simple script which uses BS to extract text from html tags. It uses lxml parser:

```
1  #
2  from bs4 import BeautifulSoup
3  #Create a markup
4  myHTML= '''
5  <!DOCTYPE html>
6  <html>
7  <body>
8  <h1>This is a Heading</h1>
9  <p>This is a paragraph.</p>
10 </body>
11 </html>'''
12 #Create a soup object from the markup using 'lxml' parser
13 soup = BeautifulSoup(myHTML, 'lxml')
14 #Use text property and prettify() method of BS object
15 print(soup.text)
16 print(soup.prettify())
```

Explanation:

Lines 4-11: Here you create a markup myHTML.

Line13: Here you create a BS object and call it soup

Line14: BS objects have a .text property which gives the text (minus the tags) of the BS object. Note that BS's text attribute will return a string stripped of any HTML tags and metadata.

Line16: BS objects have a .prettify() method. The prettify() method will create a HTML/XML parse tree from the BS object.

The screen shot of output on Jupyter is shown in Figure 23.2.

[8]See: https://developer.mozilla.org/en-US/docs/Web/HTML/Element

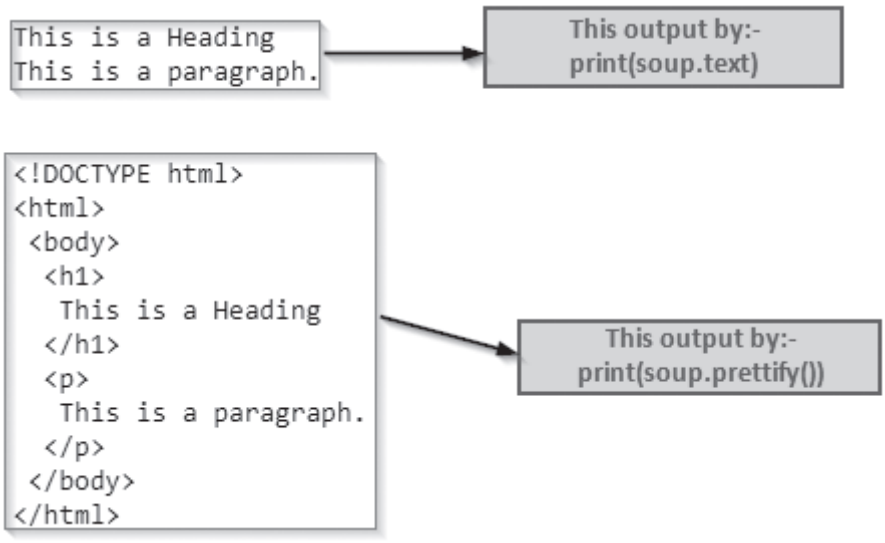

FIGURE 23.2 Screen shot on Jupyter of attribute soup.text and method soup.prettify()

BS converts an HTML document into a tree of Python objects which are basically of following four types: (1)Tag, (2) NavigableString, (3) BeautifulSoup and (4) Comment.

This book is just a brief introduction to BS, so the details of these types are not discussed here.

To explain how BS actually works in a real world program, you may write a script which uses the Google search engine to get results and then filters these results for only the url of the websites.

The script will consist of the following steps:

1. Create a User Agent. Most websites insist that as part of the API request, the user also send his User Agent (UA). UA is nothing but details of the web browser being used by the client. One can find one's user agent online here[9]. You may pass the UA as value for the headers parameter to the requests.get() method. For example, the user agent for author's browser is shown in Figure 23.3.

FIGURE 23.3 Screen shot of User Agent (UA) of the author's browser

2. Create a function which takes three parameters: (1) the "string" to be searched called searchItem, (2) the number of items to be returned by search called numbItm. It will have a default value of 20

[9]See: https://www.whoishostingthis.com/tools/user-agent/

and (3) The language of the search, i.e.,"lan" which will have a default value of "en" which is code for English. (If you want results in some other language, say Hindi, you could replace it with "hn". (You can see the language codes for all languages at this footnote[10].

3. You need to understand how Google accepts a search string. There are 2 important things to note.
   (a) First when you type something in Google search engine box, the blank spaces are replaced by "+". So you must modify the search string given by the user and replace all " " (i.e., a blank space) with "+". To do this you should use the .replace(' ', '+') method of a string class which you have studied before.
   (b) Second you need to understand how Google sends request from the client page to its server. It is quite complicated, but you need to focus only on three items: (a) the search query string which is passed to Google with "q" (q stands for query), (b) the number of results returned which is passed to the Google server by client as "num", and (c) Finally the language code which is passed to the Google server with "hl'. So you may pass your query to google as a url like this: "https://www.google.com/search?q={}&num={}&hl={}".format(modItem, numbItm, lan). Here modItem represents the search string appropriately modified by replacing the spaces with'+'. Further numbItm represents the number of results to be returned and finally lan represents the language of results.
4. Once you have constructed the query url, you can pass it to the requests.get() method. The response of this get() will be converted to text and returned by the calling function. Note that the response of the get() method is a Response object. However BS doesn't need the Response object, rather it needs the text of the Response object. So you should return the "text attribute" of the response object.
5. Now you may use BS to extract all links using <a> tag. You should also use regular expression "^http://", which means all strings starting with http:// because the token "^" stands for strings beginning with. You may also use the find_all() method of BS4. Note in BS3 it used to be findAll() but now that has been deprecated. You can always look up the signature of find_all() on Jupyter by BeautifulSoup.find_all? . The result will be something like:

```
Signature: BeautifulSoup.find_all(self, name=None, attrs={}, recursive=True,
text=None, limit=None,**kwargs)
Docstring:
Extracts a list of Tag objects that match the given criteria.  You can specify
the name of the Tag and any attributes you want the Tag to have. The value of a
key-value pair in the 'attrs' map can be a string, a list of strings, a regular
expression object, or a callable that takes a string and returns whether or not
the string matches for some custom definition of 'matches'. The same is true of
the tag name.
```

Now:

The Tag object has the attrs attribute, which returns a dictionary of key-value pairs. In this case you may want to select only those tags whose key value pair are of type: {'href': http://some_address'}. So for the key you need "href" while for the value, you need a string beginning with http://. This is why you should use: find_all('a', attrs={'href': re.compile('^http://')}). This code means find all tags matching with "a" and having attribute href beginning with http://.

[10]See: https://developers.google.com/admin-sdk/directory/v1/languages

The code is as follows:

```
#1------------imports-----------
import re
import requests
from bs4 import BeautifulSoup
#2------------user agent----------
#Use https://www.whoishostingthis.com/tools/user-agent/ to find UA
UA = {'User-Agent':'Mozilla/5.0 (Windows NT 6.1) AppleWebKit/537.36 \
        (KHTML, like Gecko) Chrome/64.0.3282.186 Safari/537.36'}
#3----define a function to get results from google search--------
def getResp(searchItem, numbItm = 20, lan = 'en'):
    modItem = searchItem.replace(' ', '+') #Replace space with +
    gUrl = ('https://www.google.com/search?q={}&num={}&hl={}'
            .format(modItem, numbItm, lan))
    try:
        response = requests.get(gUrl, headers=UA)
        return response.text
    except requests.exceptions.RequestException as e:
        print('Something went wrong', e)
        return None
#4--call function with search 'rivers of india', 20 results in english---
resp = getResp('rivers of India', 20, 'en')
soup = BeautifulSoup(resp, 'html5lib')
for link in soup.find_all('a', attrs={'href': re.compile("^http://")}):
    print(link.get('href'))
```

Line7: Here you construct the User agent UA. Note that not all servers insist upon presence of UA. Even on Google, the code might work without UA for a few queries, but if you try to repeat queries without UA, the server might block you. (***Note:*** At times it is possible that the server may block you even with a UA. This is because such scripts might be seen as "malicious" by the server.)

23.5 SOCKET PROGRAMMING IN PYTHON

23.5.1 TCP/UDP

When using sockets, two options, i.e., the two types of protocols are considered: (1) UDP (User Datagram Protocol), and (2) TCP (Transmission Control Protocol).

In this book UDP is not discussed in details. In brief, the differences between the two are shown in Table 23.1.

TABLE 23.1 Differences between the UDP and TCP protocol

Sl. No.	UDP	TCP
1	Unreliable. There is no concept of either acknowledgement or timeout	Reliable. It monitors message transmission
2	Data is not ordered. The order of data is in the order of receipt by client.	Ordered. Buffering is provided to ensure that data is received in same order it is transmitted
3	Light overhead	Heavy overhead
4	Higher speed	Lower speed

Meaning/Definition/Concepts

Python offers two different types of API libraries for socket programming:

- Socket library at lower end. This library can be used library to implement both client and server modules. It can be used for both UDP (i.e., connectionless) and TCP (i.e., connection-oriented) network protocols.
- Libraries like ftplib and httplib at higher end. They can interact with application-level network protocols like FTP and HTTP.

23.5.2 Sockets

Some relevant aspects of sockets are as follows:

- They are the endpoints of a communication channel which may send data in both directions.
- Sockets can do communication "within a particular process", or "between processes but on same machine" or even "between processes on different machines".
- There can be "client" sockets as well as "server" sockets. The client application (like a web browser) uses "client" sockets. The server on the other hand may use both client as well as server sockets.
- When one talks of sockets, one means two different things namely: address families and socket types:
  - The "address families" of sockets control what is known as the OSI network layer protocol
  - The "socket types" control what is known as the OSI transport layer protocol

23.5.3 Socket Address Family

There are a number of socket families. Details of all the family types are not discussed. However, there are two important families, you need to know: (1) UNIX domain sockets, and (2) TCP/IP sockets families. A UNIX socket is for data exchange between two different processes on the same machine, i.e., on the same operating system. On the other hand, the TCP/IP socket is for exchange between processes which could be on same or different machines.

Some common address families are:

AF_NET: This is the most common and used for IPv4 address. IPv4 addresses are made up of four octal numbers (each from 0 to 256 decimal) separated by dots.

AF_INET6: This is for using the IPv6 Internet addressing system. IPv6 is for future and supports 128-bit addresses.

AF_UNIX: This socket family is for Unix Domain Sockets (UDS). In brief UDS is a protocol for interprocess communication for POSIX systems. (They are not used in this book.)

In current example only using AF_NET is used.

Socket types (SOCK_DGRM and SOCK_STREAM)

There are many different types of sockets. But you should concern yourself with only 2, i.e., SOCK_DGRM and SOCK_STREAM. Table 23.2 indicates the two sockets.

TABLE 23.2 Differences between SOCK_STREAM and SOCK_DGRAM

	SOCK_STREAM	SOCK_DGRAM
Protocol	TCP	UDP
Reliability	Reliable	Unreliable
Connection	Connection-oriented	Connectionless
Usage	Commonly used	Rarely used

Note: TCP is acronym for Transmission Control Protocol. It is called a "reliable connection based" protocol. The term "connection oriented" means that a connection needs to be established before transmission can be done. It is used by applications like web browsers. On the other hand, UDP is a "connectionless protocol". The "reliability" part if required needs to be handled by the application layer. UDP is a faster transmission protocol than TCP. For data that requires reliability, data integrity an sequential transmission, TCP should be used. However, if speed is more important than data reliability then UDP may be preferred. For example, an application for "online gaming" needs to be fast and not so reliable. So UDP would be better for such protocol. However, most ordinary applications to be used through a browser need to be reliable and hence use TCP.

In the examples given here, only SOCK_STREAM is used.

23.5.4 Socket Library

One typically begins a socket script in Python by importing the socket library. You can divide the functionalities of socket module into following categories:

1. Functions of the socket class to create socket objects. This category consists of those functions which are used to "Create" socket objects. The most important function of this class is socket.socket() whose complete signature is: `socket.socket`(*family=AF_INET*, *type=SOCK_STREAM*, *proto=0*, *fileno=None*). Notice that here you are using class function so you must use socket.socket()
2. Functions of the socket class to do "other things". This category consists of those socket class functions which are used to do "other things". One example of this type of socket class function is `socket.gethostbyname(hostname)`[11]. This function converts a host name to an IPv4 address format. The IPv4 address is returned by the function as a string, such as "172.217.9.228". However, if the host name is an IPv4 address itself, then it is returned unchanged. Note here also you need to use socket.func_name() where func_name is the name of the function of the socket class being used.
3. Methods of a socket object. These are the methods of a socket object. Please note the difference between a socket class and a socket object. The socket class becomes available to you once you import the socket module. But a socket object becomes available to you only when you "create" a socket object by calling one of the socket creation functions like `socket.socket(parameters)`. Further the object is the return value of the socket.socket(parameters) function. Suppose my_socket is a socket object created as a "return" of a socket creation function like say my_socket = socket.socket(parameters). Then some examples of methods of socket object (not socket class) are my_socket.accept(), my_socket.bind(parameters), my_socket.close(parameters), my_socket.connect(parameters), my_socket.getsockopt(parameters), etc.
4. Constants of the socket module. The socket module has a number of constants. Many of these constants are used as "parameters" to socket class functions or socket object methods. By convention module constants in Python are words consisting of all capital letters and wherever needed they are separated by underscore. For example, the socket.AF_INET is a constant of socket class which specifies IPv4 protocol. Similarly socket.SOCK_STREAM specifies TCP protocol.
5. Exceptions of the socket module. In Python 2.x, the exceptions were available in the socket module as socket.error, socket.timeout, etc. But in Python 3.3 it is better to use OSError as the exception class for socket. So you can use both socket.error or OSError but this is the preferred method in Python 3.x

[11]See: https://docs.python.org/3/library/socket.html#socket.gethostbyname_ex

Figure 23.4 shows the various components of the socket module.

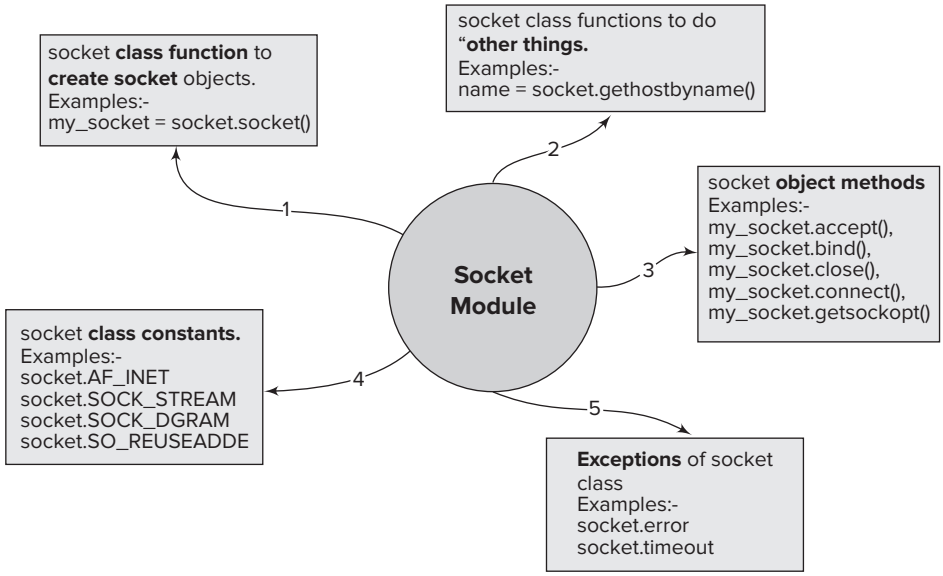

FIGURE 23.4 Important classes and methods of the socket module

The above concepts will become clear from the following discussion.

One of the most important methods of the socket class is the socket method.

Signature of socket.socket:

```
sock_obj = socket.socket(family=<AddressFamily.AF_INET:2>,
type=<SocketKind.SOCK_STREAM:1>, proto=0, fileno=None)
```

Note sock_obj is an arbitrary name for the socket object. You can use any valid Python name for the socket returned.

In simplified form, the above signature can be written as:

```
sock_obj = socket.socket(socket_family, socket_type, proto = 0)

Parameters:-
(1) socket_family: This parameter indicates the family of protocols used as the
transport mechanism. It can have either of the two values.
    - AF_UNIX, or
    - AF_INET (IP version 4 or IPv4).
(2) socket_type: It specifies the type of data exchange between the two end-
points. It can have following values.
    - SOCK_STREAM (for TCP ie connection-oriented protocols), or
    - SOCK_DGRAM (for UDP ie connectionless protocols).
(3) proto: This field defines the protocol. This field is typically left or else
given a value of 0.
Return type:-
sock_obj:- This is the return value of the method socket.socket(). It returns a
socket. You can use this socket object on both server side as well as client
side.
```

The following script shows use of socket.socket() method:

```
1  import socket
2  my_soc = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
3  print(type(my_soc))
4  print(my_soc)
5  #Output
6  <class 'socket.socket'>
7  <socket.socket fd=208, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM,
8  proto=0>
```

Line2: This is the socket.socket method. First note that socket is a module and socket.socket() is a method of the socket class. This is why you need to use socket.socket(parameters) and not just a single word socket. Now as mentioned above, this method has three important parameters. The first is the family, the second is the type and the third is the protocol

Some common methods of socket are discussed in the following sections.

23.5.5 socket.gethostbyname(host)

This is a function of the socket class. It gets the corresponding numerical network address of a host name. The following code on Jupyter shows how to use this method:

```
1  import socket
2  def getAddress(host):
3      try:
4          myAddress = socket.gethostbyname(host)
5          print(myAddress)
6      except socket.error as e:
7          print('Error->', e)
8  #Call the function
9  getAddress('google.com')
10 getAddress('xyz.abc') # Give a non existent host name to generate error
11 #Output
12 172.217.166.46
13 Error-> [Errno 11004] getaddrinfo failed
```

Explanation:

Lines 2-7: This is the definition of a function which takes a host name and prints its network address. In line3 you use the gethostbyname(host) method which returns a network address corresponding to a host name. You may wrap the call in a try except block to catch any errors. The socket.error defines the errors which may be thrown when socket calls do not execute properly.

Lines 9-10: In line9 you call the function with a valid address and in line10, on purpose, you call the function with an invalid hostname to generate an error.

You may first create a simple port scanner. A port scanner is a simple script which will scan a particular host for open ports.

```
1  import socket
2  try:
3      h_url = input("Enter the url of host to be scanned: ") # Example:- google.com
4      h_ip = socket.gethostbyname(h_url)     # Resolve h_url to IPv4 address
5      print(h_ip)      # Print IP address of host
6  except:
7      print("Wrong hostname... quitting")
8      quit()
9
10 for count in range(3): #Give 3 opportunities
11     try:
12         h_port = int(input("Enter the port: "))       # Enter the port number
13         sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
14         sock.connect((h_ip, h_port))
15         print("Port {}: is open" .format(h_port))
16         sock.close()
17     except:
18         print("Port {}: is closed" .format(h_port))
19
20 print("Port Scanning over... quitting")
21
```

```
22 #Output
23 Enter the url of host to be scanned: wikipedia.com
24 91.198.174.192
25 Enter the port: 80
26 Port 80: is open
27 Enter the port: 20
28 Port 20: is closed
29 Enter the port: 443       #443 is https port for secure communication
30 Port 443: is open
31 Port Scanning over... quitting
```

Explanation:

Lines 2-9: This is a try-except block to input the host name by the user. If something goes wrong say wrong input by user, the script will quit gracefully. It is always a good idea to wrap user input in a try-except block.

Lines 11-19: There is a for loop which runs three times. Inside the for loop there is a try-except block.

Line13: Here you ask user to give a port number.

Line14: Here you create a socket object named sock.

Line15: You may connect the socket object to the given IP address and the given port. If this code fails, it will throw an exception, which will be caught by the except block. So if the user gives a port number which is closed, then there will be an exception and line 19 will be printed.

23.5.6 Creating a Simple Server and a Client Using Socket Module

Now you may create a simple server and a simple client in Python using the socket module.

- Steps in setting up a server using socket library:
  - Create socket on server side using socket.socket(parameters) function of socket class. Provide parameters like AF_INET (For IPv4) and SOCK_STREAM (For TCP). ***Note:*** SOCK_STREAM means that it is a TCP socket. SOCK_DGRAM means that it is a UDP socket.
  - Set certain socket options like SOL_SOCKET (The option SOL_SOCKET is used to specify the argument "level" of setsockopt() function as SOL_SOCKET.[12]) and SO_REUSEADDR (For ensuring that the port does not get stuck in TIME_WAIT state)[13].
  - Bind the socket created to the hostname and port specified. After this step, the server will get linked to the IP address of the machine on which the server is running and will also get linked to a port on which it will listen to incoming requests from clients.
  - Put the server created in listen mode by using the listen(parameter) method of socket object. Here there is one parameter which specifies how many incoming connections, the server should listen to.
  - Create an infinite loop. Inside this loop the server accepts the incoming connection by using accept() method of socket object. Note that the "return value" of the accept method creates another socket object. So now you have 2 socket objects, one created by the socket() function of the socket module and the other by the accept() method of socket object. Note that the socket created by the socket() function is a "passive socket" in the sense that it only "carries with it" the IP address and the port of the server. It does not do any actual communication with the client. However, the "active socket" created by the return value of the accept() method is the socket which does the "actual talking" to the client. So it may be noted that on the server side, there are two types of sockets. This is due to the fact that the standard POSIX interface[14] to TCP actually has two completely different kinds of sockets, namely: (1) "passive" listening sockets, and (2) active "connected" sockets. Note that the "active socket" created on the server side has a tuple of four items, i.e., (server_IP, server_PORT, client_IP, client_PORT). From this tuple of 4 items, the OS of the server is able to route the incoming TCP packets to the correct socket on server side.
  - Use the "active socket" created to send and receive data from the client.
- Steps in setting up a client using socket library:
  - Create socket on server side using socket.socket(parameters) function of socket class. Provide parameters like AF_INET (for IPv4) and SOCK_STREAM (for TCP).
  - Connect to the server giving the IP address of the server and also the port number on which the server is listening. Note the difference between the steps on server side and client side. On server side you may use bind() while on client side you may use connect()

---

[12]Note that one of the signatures of the setsockopt(parameters) method of a sock object is:

setsockopt(level, option, value: int)

Here the first argument is level and for this you may specify SOL_SOCKET to indicate that you are operating at socket level. Other options are possible but details are beyond the scope of this book.

[13]Note: If one tries to rapidly create multiple sockets using the same IP:port address pair, one gets an error of type "address already in use". This is because the earlier socket has been fully released. Even if a client closes a connection, the server generally likes to wait for some time before closing the connection, so the address will be in use even if the client closes the connection. Setting the SO_REUSEADDR flag will ensure that the server does not wait for some time after the client has closed but releases that particular IP:port address pair.

[14]POSIX specifications evolved from Berkley sockets aka BSD sockets.

- After connection is established, the socket can be used to send and receive data.
- Finally when communication is done, close the socket so that the server knows that the client has finished communication and after a time interval, the server can release the connection and allow other clients to connect.

The overall scenario is shown in Figure 23.5.

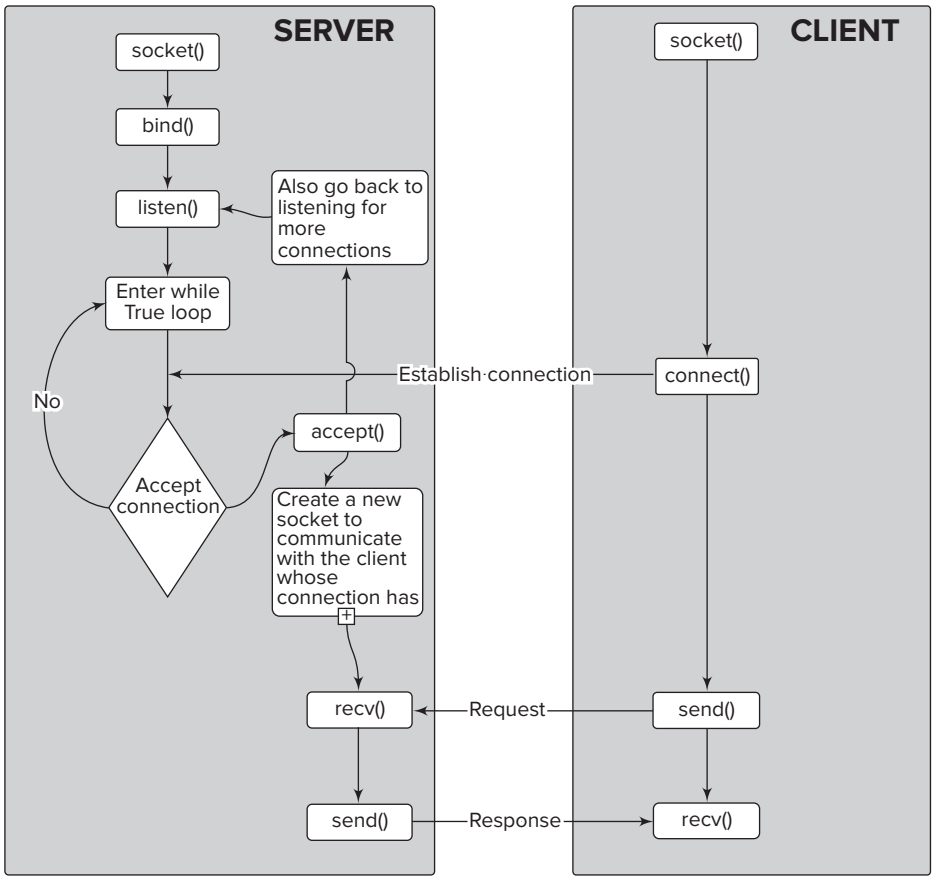

FIGURE 23.5 Connection between a client and a server

A server and a client are created on local machine using sockets. But only TCP and not UDP is used because TCP has become the defacto protocol. The general flow of the program is as follows:

```
1  import socket
2  import time
3  # get local machine name
4  HOST = socket.gethostname()
5  PORT = 9999#For port number
6  CLIENTS = 5#For max number of clients permitted
7
```

```
# create a socket object
try:
    #create an INET, STREAMing socket
    serv_socket = socket.socket(
            socket.AF_INET, socket.SOCK_STREAM) #SOCK_STREAM-> TCP
    #setsockopt must be done before bind
    serv_socket.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    # bind to host and port.
    serv_socket.bind((HOST, PORT))
    # queue up to 5 requests
    serv_socket.listen(CLIENTS)
except OSError as e:
    print('Error->', e)

while True:
    # establish a connection
    (client_socket,addr) = serv_socket.accept()
    print("Connected to %s" % str(addr))
    currentTime = time.ctime(time.time()) + "\r\n"
    client_socket.send(currentTime.encode('ascii'))
    client_socket.close()
```

```
# client.py
import socket

# get local machine name
HOST = socket.gethostname()
print('Host name->', HOST)
PORT = 9999
try:
    # create an INET, STREAMing socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # connection to hostname on the port.
    s.connect((HOST, PORT))
    # Receive no more than 1024 bytes
    tm = s.recv(1024)
    #Close the port
    s.close()

    print("The time got from the server is %s" % tm.decode('ascii'))
except OSError as e:
    print('Error->', e)

```

(***Note:*** If you run the server script on one instance of Jupyter notebook and the client script on another instance, then the script should run)

When the two scripts were executed on two instances of Jupyter on the author's computer, the following was the output on server side.

```
Connected to-> ('192.168.31.123', 57741)
Connected to-> ('192.168.31.123', 57742)
Connected to-> ('192.168.31.123', 57743)
Connected to-> ('192.168.31.123', 57749)
Connected to-> ('192.168.31.123', 57751)
```

The output on client side was:

```
Host name-> DS-PC
The time got from the server is Fri Nov 16 10:17:23 2018
```

23.5.7 Sending and Receiving Data (Data exchange between server and client) Using Sockets

- One of the most important issues to remember while using sockets for sending/receiving messages is that: Sockets are byte streams not message streams. So what is the difference? The difference is the "Message boundary". A message stream will keep track of the message boundary, i.e., where the message ends but a byte stream does not keep track of the message boundary.
- Another important point to note is that in TCP, the message is "buffered". Why? Because TCP is a reliable protocol and if the message is lost, then the buffer will be used to retransmit it. This has very important implications. So when you use a socket object method like `send(data)`, it means that the data has been sent to the TCP buffer, it does not necessarily mean that the data has been actually transmitted. All it means is that the data has been copied into the input buffer. (However, UDP socket packets are not buffered). The situation is somewhat as shown in Figure 23.6.

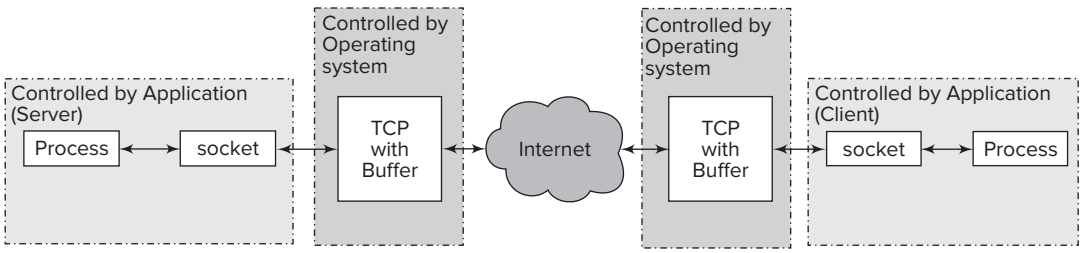

FIGURE 23.6 Interaction between a server and a client using the TCP protocol

For sending data using sockets, you generally have two methods available to us, namely: send() and sendall().

Note that there is no guarantee that the send() method of socket will send all of the data you give to it. The send() method however will return the number of bytes that were actually sent and now it is upto your application as to how it will retransmit the unsent data.

However, Python provides a method called sendall() which ensures that all of your data is sent before returning. socket.send() is a low-level method. It may or may not send all the bytes provided to it, but it will return the number of bytes sent.

On the other hand, socket.sendall() is a high level method. It will send all the data given to it, unless an exception occurs. After it has sent all the data, it will return a value of None. This method is much easier to use. The only problem with this method is that suppose you use this method to send data and an exception occurs, then there is no way of knowing how much data was sent.

Similarly for receiving data, you have a function socket.recv(). Again the problem with recv() is that it will return a 0. This means the other side has closed/or about to close the connection.

You may first write a simple application which sends a message to the Wikipedia server and gets a message back. A *"GET / HTTP/1.0\r\n\r\n"*.request is used:

```
1  # client.py
2  import socket
3
4  # get local machine name
5  HOST = "www.wikipedia.org"
6  PORT = 80
7  try:
8      s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
9      s.connect((HOST, PORT))
10     # send request via http GET method
11     s.send("GET / HTTP/1.0\r\n\r\n".encode('ascii'))
12     print(s)
13     # Receive no more than 2000 bytes
14     data_got = s.recv(2000)
15     print('Bytes received->',len(data_got))  #Print number of bytes received
16     print('First few bytes->',data_got[:10])
17     print('Last few bytes->',data_got[-10:-1])
18     #Close the port
19     s.close()
20
21 except OSError as e:
22     print('Error->', e)
```

```
22 #Output
23 <socket.socket fd=336, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM,
24 proto=0, laddr=('172.16.3.4', 50136), raddr=('91.198.174.192', 80)>
25 Bytes received-> 1054
26 First few bytes-> b'HTTP/1.1 2'
27 Last few bytes-> b' close\r\n\r'
```

Line11: Here you have used an http GET request which is one of the most common requests used in APIs and websites. GET is used to fetch data from a server. Note that a GET request only requests data and doesn't do any modification. Hence, it is generally considered a safe and idempotent method.

The following creates a client which opens a file location and sends the entire file data to a server. It also create a server which allows a client to connect and reads incoming data and stores it in a file.

The client code is:

```
# client.py
import socket

# get local machine name
HOST = "www.wikipedia.org"
PORT = 80
try:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((HOST, PORT))
    # send request via http GET method
    s.send("GET / HTTP/1.0\r\n\r\n".encode('ascii'))
    print(s)
    # Receive no more than 2000 bytes
    data_got = s.recv(2000)
    print('Bytes received->',len(data_got))  #Print number of bytes received
    print('First few bytes->',data_got[:10])
    print('Last few bytes->',data_got[-10:-1])
    #Close the port
    s.close()

except OSError as e:
    print('Error->', e)
```

```
#Output
<socket.socket fd=336, family=AddressFamily.AF_INET, type=SocketKind.SOCK_STREAM,
proto=0, laddr=('172.16.3.4', 50136), raddr=('91.198.174.192', 80)>
Bytes received-> 1054
First few bytes-> b'HTTP/1.1 2'
Last few bytes-> b' close\r\n\r'
```

The server code is:

```
#server
#gets file from client
import socket
import sys
HOST = socket.gethostname()
PORT = 4321

try:
    s1 = socket.socket(socket.AF_INET, socket.SOCK_STREAM)#s1 is static socket
    s1.bind((HOST, PORT))
    s1.listen(5)
    print("Server is listening ..")
    #Outer while loop is for creating active socket
    while True:
        (s2, addr) = s1.accept()#s2 is dynamic socket
        print("Client connected->", addr)
```

```
        # select file location to save incoming data
        fObj = open(r"C:\Python34\myScripts\abc.txt", "wb")
        #Inner while loop is for reading incoming data
        while True:
            f_data = s2.recv(4096)
            if not f_data:
                break
            # write data to file
            fObj.write(f_data)
        #Finished reading, back to outer while loop
        fObj.close()
        print("File Downloaded")

        # close s2ection
        s2.close()
        print("Client disconnected")
        sys.exit(0)
except IOError as e:
    print("Error->", e)
```

23.6 ftplib AND smtplib

This section discusses twoPython modules namely ftplib and smtplib. Ftplib is for FTP (File Transfer Protocol) while the smtplib is a module for sending mail to any system connected on Internet and having a SMTP listener daemon.

23.6.1 FTP and ftplib

The File Transfer Protocol (FTP) is used for transfer of files between a client and server on a network. FTP was not designed to be a secure protocol, and has many security weaknesses.

It is possible to create both ftp servers as well as ftp clients. But in this book only creation of a ftp client (not ftp server) is discussed. In this exercise an ftp client is created and then this client is used to connect to one of the public ftp service available to prove that the script is actually working.

For this a well-known module ftplib is used. You may use the ftplib to connect to some online ftp service. One such service is offered by ftp://ftp.gnu.org/. Screen shot of this website is shown in Figure 23.7.

You may access this website using an ftp client. Library ftplib is used.

Certain noteworthy points regarding ftplib module are as follows:

- FTP is a class in the ftplib module. The constructor for this class is:
  class ftplib.FTP(host='', user='', passwd='', acct='', timeout=None, source_address=None)
  In this example code you may download from a known ftp server site, i.e.,ftp.gnu.org. Note that the return value of this constructor call is an object of FTP class. So if you have:
  my_ftp = ftplib.FTP("ftp_address"), then the object my_ftp will be an object of class FTP and will point to the ftp server at the url 'ftp_address'
- Once you have created a FTP object say my_ftp, then you can use my_ftp.login("username", "password") to connect to the ftp service.

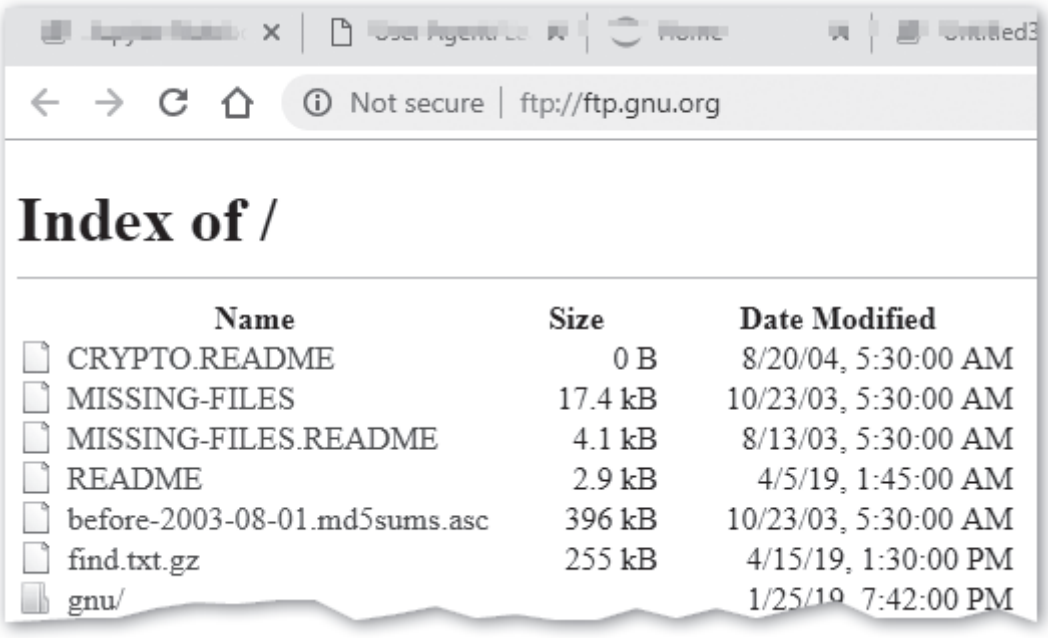

FIGURE 23.7 Screen shot of web page ftp://ftp.gnu.org/

- You can use the my_ftp.dir() method to get the list of the directory at the ftp server.
- You can download a file from the ftp server using the ftp object say my_ftp by using a method ftp.retrbinary(). The signature for retrbinary method is:
  - FTP.retrbinary(cmd, callback, blocksize=8192, rest=None)
  - Here cmd should be a RETR command[15]: "RETR filename". What does this mean? It means that you have to use the word RETR followed by the name of the file on the ftp server that you want to download. A RETR request asks the server to send the contents of a file over the data connection already established by the client. (Note in Python you can always concatenate two strings so you can have "RETR" + "ftp_file". But note that there should be at least one blank space between 'RETR' and "ftp_file".)
  - Further callback is a callback function which is called for each block of data received, with a single bytes argument giving the data block. In this code example, you may create a file object with the command say local_f = open("local_file_name", "mode") and then use the local_f.write as a callback function. (Note since you have to write to this file, the mode must be write not read).
  - Finally the optional blocksize argument indicates the maximum chunk size to read on the socket object.

Code is as follows:

```
1 from ftplib import FTP
2 try:
```

[15]RETR is one of the many ftp commands that can be sent to a ftp server by a client. For full list of ftp commands see: https://tools.ietf.org/html/rfc959

```
    with FTP("ftp.gnu.org") as ftp:
        login_resp = ftp.login("anonymous", 'abc@xyz.com')
        # Just print first 50 characters of response
        print('response->',login_resp[:50])
        ftp.dir()# Get directory listing of ftp server
        #Open file on client side to store data downloaded from ftp
        # Put your file path here
        local_f = open(r"C:\Python34\myScripts\abc.txt", 'wb')
        ftp.retrbinary("RETR " + "README", # File opened on server is README
                       callback = local_f.write, # callback writes to file
                       blocksize= 1094 )
        print("download complete")
        ftp.close()
        print("disconnected......")
except Exception as e:
    print('something wrong', e)
```

```
#Output. . .
response-> 230-NOTICE (Updated October 13 2017):
230-
230-Bec
lrwxrwxrwx    1 0        0               8 Aug 20  2004 CRYPTO.README -> .message
-rw-r--r--    1 0        0           17864 Oct 23  2003 MISSING-FILES
-rw-r--r--    1 0        0            4178 Aug 13  2003 MISSING-FILES.README
- - - Rest of output truncated to save space - - -
download complete
disconnected......
```

The above script does two things.

- First it outputs the details of the directory of the ftp server.
- Second, it copies the data in the README file on the ftp server to a local file, whose full path name is: *'C:\Python34\myScripts\abc.txt'*.

23.6.2 smtplib Module

You may use the smtplib module to demonstrate how to send an email from a gmail account. For running this code, you may have to change the settings of your gmail account to permit less secure apps to access it. See this link[16].

```
import smtplib

server = smtplib.SMTP('smtp.gmail.com', 587)
server.starttls()
server.login("source_mail_ID", "password")

msg = "Bla...Bla...Bla"
server.sendmail("source_mail_ID","destination_mail_ID", msg)
server.quit()
```

[16]https://support.google.com/accounts/answer/6010255?hl=en

Line3: Note "smtp.gmail.com" is the url of the SMTP server of gmail.Note port 587 is the standard port used by an SMTP client to send mail to SMTP server.

Line8: Note that the destination_mail_id can be a list of IDs also. For example, you could do something like this: destination_mail_id = ['id1@xyz.com, id2@gmail.com, id3@hotmail.com ']

The above code is pretty simple and self-explanatory. However, its functionality is also highly limited. It is of course possible to create applications which can do more complicated operations like sending attachments, etc. but they are not covered here.

Note that the above example code simply sends a very small message. But an email message can be very complex consisting of multiple parts and attachments. To create complex messages and sending them, you can use a module in Python called email.

Overview of the email package is shown in Figure 23.8.

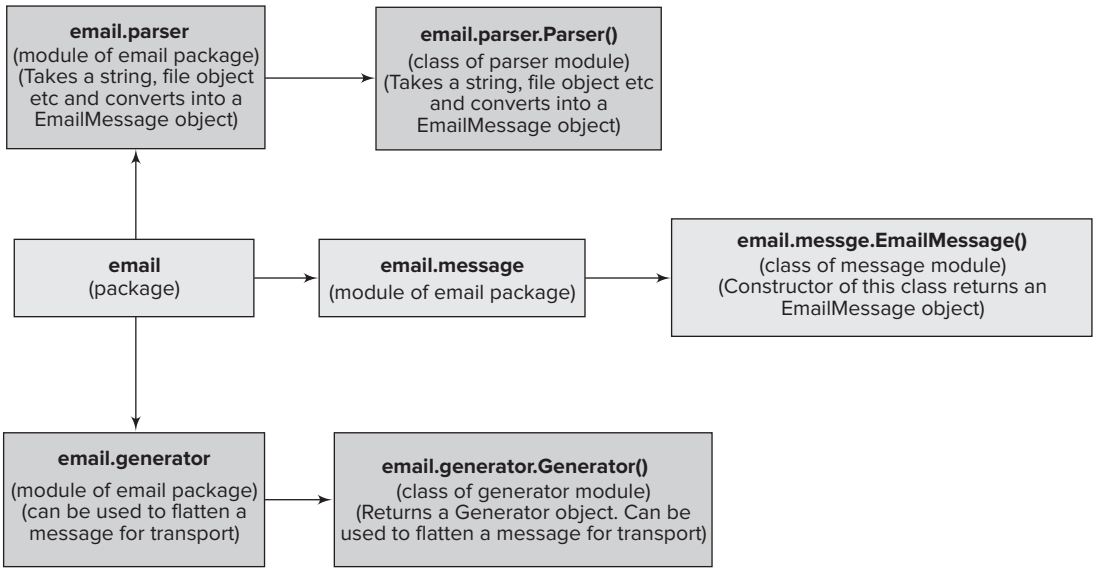

FIGURE 23.8 Layout of the Python email module

```
1  # Import smtplib for sending a file
2  import smtplib
3  # Import email modules for message construction only
4  from email.message import EmailMessage
5
6  # Open file and create EmailMessage object from it.
7  with open(r"C:\Python34\myScripts\abc.txt", 'w+') as fObj:
8      # Create a  message
9      msg = EmailMessage()
10     #use set_content()
11     msg.set_content(fObj.read())
12
13 me = "sender_mail_ID"
14 you = "receipent_mail_ID"
```

```
15 msg['Subject'] = 'file abc.txt'
16 msg['From'] = me
17 msg['To'] = you
18 smtp_conn = smtplib.SMTP('smtp.gmail.com', 587)
19 smtp_conn.starttls()
20 smtp_conn.login(me, "password")
21 smtp_conn.send_message(msg)
22 smtp_conn.quit()
```

Lines7-11: Here you may first open a file to be sent via SMTP. Then you construct a message whose contents are from a file abc.txt. Then you may use set_content() method to attach data from file to the msg object. For further details of the Python email package, follow the link below[17].

Line9: Here you create an EmailMessage object and call it msg.

Line11: You may use set_content() method of EmailMessage class to set the contents of the msg object.

Conceptual Questions

1. What is the whois module of Python?
2. What is the "object type" returned by a whois.whois() method?
3. What is the requests module in Python? What are the four important components of a request?
4. Explain the four types of requests, i.e., GET, POST, PUT and DELETE.
5. What does the method requests.get() do?
6. Discuss the signatures of the seven common types of request methods. (i.e., get, options, head, post, put, patch and delete).
7. What is the difference between requests.get() method and requests.request('GET') method?
   (Ans: They do the same thing. While requests.get() is only for getting, requests.request() may be used for other things also.)
8. What is the difference between requests.get() and requests.Session()?
9. What are cookies? How are they represented in Python?
10. What are SSL certificates?
11. What is the OAuth2.0 protocol? Does the requests module have provisions to handle OAuth2.0 protocol?
12. What is the tweepy library? Which are the four secret strings you must have in order to use the library?
13. What is the difference between RESTful and Streaming API?
14. On social media site twitter, what are home_timeline(0 and user_timeline?
15. What is the BeautifulSoup4 module? While using BeautifulSoup, you generally do an import of type: from bs4 import BeautifulSoup .Why?
16. Which is the default parser in BS? Name some other common parsers.
17. Discuss the .text property of a BS object.

[17]See:- https://docs.python.org/3/library/email.html#module-email

18. What is a User Agent (UA)? How can you get the UA of your browser? Do all websites insist upon getting the UA of the client?
19. What is Certifi?
20. What is the advantage of using a Session() object? Do cookies persist when using Session() object?
21. When you make a request like say request.get(), then what are you sending to the server and what do you get back from the server? (***Hint:*** You send a Request object to server and you get back a Response object from the server)
22. How can you get the header the server sent back to client? (***Hint:*** use r.header where r is the Response got from server).
23. What do the terms UDP and TCP stand for? Give major differences between the two protocols.
24. What is a socket? Which are the common socket address families?
25. What does the constant AF_INET of the socket class indicate?
26. What do the constants SOCK_STREAM and SOCK_DGRAM indicate?
27. Write a script which creates an INET STREAMing socket.
28. What is a GET request? Why is it generally considered safe?
29. What are the weaknesses of the FTP protocol?
30. What is the smtplib module of Python? What is the standard port used by SMTP client to send mail to an SMTP server?

Exercise

1. Examine the following code:

```
1 import requests
2 tw_url = 'https://twitter.com/'
3 req1 = requests.get(tw_url)
4 print('req1 type->', type(req1))
5 print(req1.headers)
6 print(req1.request.headers)
```

What type of object is req1?
What is the difference between req1.headers and req1.request.headers? Which are the headers from server to client (i.e., response header)? Which are the headers from client to server (i.e., Request header)? Does the req1 object in above code contain Request header or Response header or both?

Beyond Textbook

1. While discussing the requests library streaming upload/download were not discussed. How can you do streaming download using requests.get(). (***Hint:*** Look at the stream parameter and set it to True). How can you use the iter_lines() method of the Response object to iterate over the data got from server?

2. Using the "auth" optional parameter in requests.request(method, url, **kwargs)
   In the text book, while discussing the requests.request(method, url, **kwargs) method it was seen that one of the **kwargs is "auth" which needs a tuple of type (user_name, pass_word).
   This exercise shows how to use the "auth" parameter of a Request object.
   Steps in exercise are:
   - The website used is httpbin.org. This is maintained by Kenneth Reitz, the author of requests library. (You can check out this site by using the url httpbin.org)
   - The endpoint used for this exercise is: https://httpbin.org/basic-auth/username/password. So if you follow this link, you will get a web page which looks somewhat like as shown in Figure 23.9.

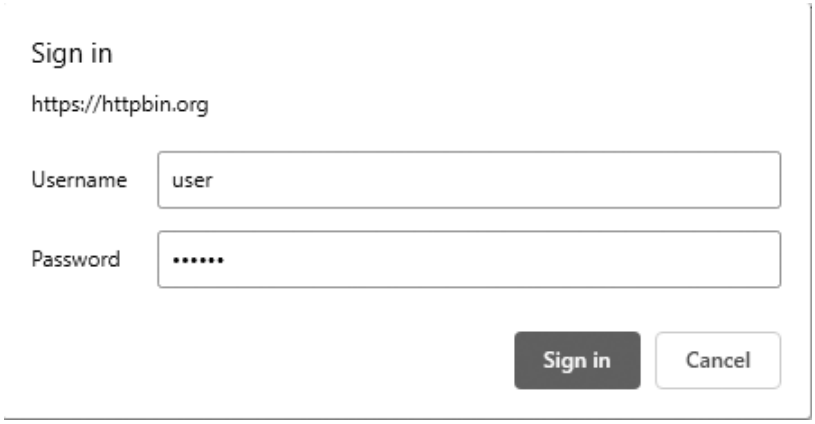

FIGURE 23.9 Screen shot of https://httpbin.org/basic-auth/username/password

   - The value for Username is username and that for password is password. So if you pass these two values for your "auth" parameter, you will be able to access the website using the "auth" parameter.

This is shown in the following example code:

```
1  import requests
2  # The endpoint for authentication on httpbin is given below
3  my_url = "https://httpbin.org/basic-auth/username/password"
4  # The endpoint accepts username = username and password = password
5  user_name = "username"
6  pass_word = "password"
7  # Create a tuple of user_name and password to give to auth parameter
8  auth_tup = (user_name, pass_word)
9
10 resp_obj = requests.request(method = 'GET',  # Method is GET
11                            url = my_url,    # url of endpoint
12                            auth=auth_tup)   # parameter to auth is a tuple
13
14 # Convert JSON to dict and print
15 print('Response from httpbin.org->', resp_obj.json())
16 # OUTPUT
17 Response from httpbin.org-> {'authenticated': True, 'user': 'username'}
```

3. A common issue while downloading files from Internet is the need to verify its integrity. One common method is to provide the md5 hash of the file. Then one needs to download the file and get its md5 hash value and compare it with the md5 value provided.
   A very useful library called hashlib is provided in Python 3[18]. In fact this library provides constructors for a number of different hashing algorithms like sha1(), sha224(), sha256(), sha384(), sha512(), blake2b(), blake2s() and also md5(). The task here is to study the hashlib library. Do the following:
   - The hashlib library has an attribute algorithm_guaranteed. This attribute provides a tuple of all algorithms provided by the library. Use this attribute to get all hashing algorithms implemented by the library.
   - You can create hash value of a string by one of the algorithms using the constructor as shown in the following script:

```
1  import hashlib
2  # Give the name of algorithm to the new() method
3  # The return is a hash object
4  hash_obj = hashlib.new('algorithm_name')
5  # give the string to be hashed as parameter
6  # to the update() method
7  hash_obj.update('string_to_be_hashed')
8  # Use hexdigest() method to get the hex value of the hash
9  print(hash_obj.hexdigest())
```

   Note: A hashing function like md5 takes only a sequence of bytes as a parameter. So the string must be preceded by letter b.
   Now that you know how to use the hashlib library, use the md5 algorithm to get hash value of the string "We shall overcome".

ASSIGNMENT

1. **Studying details of the SSL certificates**
   The requests library by default verifies the SSL certificates. Can you use requests in a way that it ignores the SSL certificate? (This of course would be an unsafe way to use requests)
2. **Studying certs.py file and .pem file**
   Have a look at the source code of the certs.py file (Either on github or your own computer. If you open source code files on your own computer, you should be careful not to alter them in any way, since accidental alteration may corrupt the files). Discuss the following:
   - What is the certifi module?
   - What is a .pem file? Examine the cacert.pem file. Where is the cert.pem file located?
3. Studying event hooks and using response hooks of the requests module.
4. The requests module provides "event hooks[19]".
   Study the event hooks for the requests module. Write a script which uses the response hook.

[18]See: https://docs.python.org/3/library/hashlib.html

[19]For details, see: http://docs.python-requests.org/en/master/user/advanced/?highlight=ssl#event-hooks

5. **Examining cookies.py file**

   In the chapter cookies as used in the requests module were discussed. In this assignment you should do the following tasks:

   - Have a look at the source code of the cookies.py file in the requests module[20].
   - Examine the RequestCookieJar class and its methods like keys(), items(), get_dict().
   - Examine the method create_cookie() of this class and create a "supercookie".

6. **Studying authentication protocols in the auth.py file**

   Another important issue which has not been discussed in the text book is authentication protocol followed in a request-response session.

   A number of authentication schemes are possible. The requests module has a source file called auth.py[21]. This file provides some common authentication protocols like:

   - HTTPBasicAuth
   - HTTPProxyAuth
   - HTTPDigestAuth

   The assignment is:

   - Study the source code of auth.py file. Identify the base class AuthBase() and see that it is an abstract class and has not implemented method called __call__(). Further see that all the other authentication classes, i.e., HTTPBasicAuth, HTTPProxyAuth(), HTTPDigestAuth() are actually derived from this abstract base class and all of them implement the __call__() method.

7. Install the requests-oauth2 module and study the oauth2 protocol
8. In the chapter a very brief introduction to the tweepy module was given. The project is to do the following:

   - Study the code in api.py file available here[22]. Within this file to see the code of API() class and its method home_timeline(). This method returns 20 status objects.
   - Write a script which uses the tweepy module to download 200 tweets from time line of a friend. Further the Status objects got in form of a list have certain attributes. For each Status object/tweet, get thefollowing attributes and store them in a .csv file:
     - author (this is the `tweepy.models.User` instance of the author of the `tweet`),
     - coordinates (this is a dictionary of coordinates in the GeoJSON format),
     - created_at (this attribute gives the creation time of the Statusobject/tweet), and
     - id (this is a unique id of the Status object/tweet)
   - Further the tweepy library has a class Cursor. By using the Cursor class, you can iterate over the Status objects collected/downloaded. (If you want to know things were done before the Cursor class was made part of tweepy library, have a look at link here[23]. This link compares the "old way, i.e., without the Cursor class" to the "new way, i.e., with the Cursor class".) Study

[20]You can see the source code for cookies.py file on github at: http://docs.python-requests.org/en/master/_modules/requests/cookies/. You can even see the file on your local computer. On author's computer this file is available at: C:\ProgramData\Anaconda3\Lib\site-packages\requests

[21]Auth.py is available at: https://github.com/requests/requests/blob/master/requests/auth.py

[22]api.py is available at: https://github.com/tweepy/tweepy/blob/master/tweepy/api.py

[23]See: http://docs.tweepy.org/en/v3.5.0/cursor_tutorial.html

the use of the Cursor class. An example code of how the Cursor class can be used is shown in the following code:

```
for status in tweepy.Cursor(api.home_timeline).items(2):
    # Process a single status
    print('text->', status.text)
```

PROJECT

In subsequent chapters, you will be doing some text processing. For this purpose you may need some text data. One way of getting text data is to download tweets.

So in this project, the following are done:

1. Create a dictionary whose keys are "types of twitter handles" and whose values are some Twitter handles which are of that type. For example you may have a key = "cricket" and handles relating to cricket, somewhat like this d = {'cricket': ''ICC', 'BCCI', 'OfficialSLC', 'England'}.
2. Use the Twitter handles to download tweets from Twitter.
3. Extract the following data from the tweets: id_str, created_at, text.
4. Save the data extracted from the tweets to a .csv file.

```
import tweepy
import csv

CONSUMER_API_KEY = 'wrxxxxxxxxxxxxxxxxxxxxv'
CONSUMER_API_SECRET = 'fxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxMR7LndeOqTw5rEMI'
ACCESS_TOKEN = '1xxxxxx3-zxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxKb'
ACCESS_TOKEN_SECRET = 'YxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxB'

auth = tweepy.OAuthHandler(
        consumer_key = CONSUMER_API_KEY,
        consumer_secret = CONSUMER_API_SECRET)
auth.set_access_token(
        key = ACCESS_TOKEN,
        secret = ACCESS_TOKEN_SECRET)

api = tweepy.API(auth_handler = auth)


d_handles = {}
d_handles['cricket'] = ['ICC', 'BCCI', 'OfficialSLC', 'England']
d_handles['python'] = ['ThePSF', 'montypython', 'PythonWeekly']

bunch_tweets = []
for key, value in d_handles.items():
    for x in range(len(value)):
        all_tweets = api.user_timeline(screen_name = value[x],
                                       count = 20)
        bunch_tweets.extend(all_tweets)
```

```
29
30 alltweets = [[tweet.id_str, tweet.created_at, tweet.text.encode('utf-8')] for
31 tweet in bunch_tweets]
32
33 path2file = r"C:\temp3\tweets_data.csv"
34 with open(path2file, 'w', newline = '') as f:
35     writer = csv.writer(f)
36     writer.writerow(['id', 'created_at', 'text'])
37     writer.writerows(alltweets) #  note writerows ie pulural
38
39 print('done')
```