Reading Data from the Web: Web Scraping & Regular Expressions

In this chapter you will learn how to read data from web servers. The urllib module allows you to download data from web servers. Typically, you will download web-pages written in HTML that were designed for a web-browser to render (draw on-screen) for a human to read. This allows you to download useful data from web-pages, such as the current temperature, sports scores, item prices from web stores, and anything else you can find on the web.

However, before learning how to do this, you should understand that some websites put legal (and ethical) restrictions on the data they provide. In some cases a website intends the data to be seen by humans, but do not want a program to download the data automatically.

Website owners have several legitimate reasons for not wanting you to download their data automatically. In some cases the website owners worry about programs overloading their servers, and costing them money. For example, a human may look at the temperature or weather forecast once or twice a day, but it would be possible to write a program that would download this same data hundreds of times a day, raising the bandwidth and hosting costs for the website owners. In other cases, the website owners have signed a contract with a data provider licensing them to display it to their (human) visitors, but do not have permission to allow others to download the data for other purposes. Many stock markets put limitations on how their pricing data can be used, especially the real-time data that is not delayed 30 minutes. In some cases website owners pay a “per-view” cost for showing or using data from a data provider, and an automated program (which would “download” or “view” the data, but is unlikely to look at online advertisements) would cost them money every time it downloaded the data, above and beyond their data/hosting costs.

Ethical Web Scraping

If you are designing a program that downloads data from a website, you should ask the website owner for written permission to retrieve the data automatically. In many cases where a website wants you to have access to their data, they will have a web-based API that makes downloading the data easier than reading it from an HTML page. Typically web-based API’s are formatted using JSON or XML to make the data returned easier to parse for a computer. (Although a computer can parse HTML, it can be complicated and take more work that you would like.)

For example, the website PaperBackSwap.com allows users to trade paperback books online. They have a web accessible search form that allows you to search for a book by title, author, or ISBN number to see if a member has that book posted for swapping. You could write a web scraper to check if a book you want is available. However, it would have to parse the HTML page that PaperBackSwap.com returns, looking for the data it wants.

PaperBackSwap.com also has a mobile version of their website that is designed for mobile phones. It features a simple interface and the pages that it returns have much less HTML, making them easier to parse. Using the mobile version of a website can make your job easier once you download the data!

However, PaperBackSwap.com also has a developer API that allows you to send search requests and get responses via XML or JSON! One big advantage of using an API is that it is an officially sanctioned method of getting access to the data. Aside from the legal protections, using an API means that the website owners and programmers know that people are using it, and won’t change the API without a lot of advance warning. If you are scraping a webpage and depending on it’s format to remain unchanged so that you can find the data you want, any time the website gets an update your web-scraper will break!

If you are unable to use an API, and your inquiries to the website operators are unanswered, you should check the sites “robot.txt” file to see if they ask that robots not crawl their website. You should also check the websites terms of service to see if they specifically prohibit the automated downloading of data.

Basic webpage download

The following four lines of python code will read the WSB TV weather webpage located at www.wsbtv.com/s/weather and convert the bytes of the webpage into a string that we can look through:

```
1  import urllib.request          #Module for reading from the web
2
3  response = urllib.request.urlopen("http://www.wsbtv.com/s/weather")
4  html = response.read()         #Gets an array of bytes!
5  strHTML = html.decode()        #Convert to a string
```

A more “correct” example actually generates a request first, and then opens that request:

```
1  import urllib.request
2
3  request = urllib.request.Request("http://www.wsbtv.com/s/weather")
4  response = urllib.request.urlopen(request)
5  the_page = response.read()
6  theText = the_page.decode()
```

In both cases, the strHTML or theText variables will point to a long piece of text (over 89 thousand characters) that contains the entire HTML for the WSB TV weather page. A small part of this text has an up to date weather forecast, including the current temperature.

The HTML code that corresponds to the temperature looks like this:

```
<span class="cmWeatherCurrentTemp">43&deg;</span><span class="...
```

Note how the “°” text immediately follows the 43 degree temperature. This is a special HTML code that draws the degree symbol. Assuming that the temperature is the first occurance of the “°” symbol, we can search for this special string to locate the (index of the end of the) temperature. But we must also figure out where the temperature starts. And because it’s possible that the high temp in Atalanta can be more than 99 degrees, and the low less than 10 degrees, we don’t know if the temperature will be one, two, or three digits in length! (Two digits is much more likely, but don’t discount the ability for a -5 degree day, which is also two characters!) To determine the size of todays temperature, we must look backwards from the “°” string until we find the greater-than symbol (>) that immediately proceeds the temperature (the end of the “” tag.)

The following code demonstrates how to load the page and find the current temp (converting it to an int).

```
1  import urllib.request
2
3
4  response = urllib.request.urlopen("http://www.wsbtv.com/s/weather")
5  html = response.read()
6  text = html.decode()                #Convert the bytes to a string.
7
8  #The temp is followed by a degree sign, which is easy to search for!
9  endOfTemp = text.find("&deg;")
10
11 #One we find the index of what is after the temp, we need to know
12 #the index of what is in front of it.
13 index = endOfTemp
14 ch = text[index]
15
16 #Move backwards from the temp until we find the end of the <span> tag...
17 while ch != ">":
18     index = index - 1
19     ch = text[index]
20
21
22 #Add one to get the first digit of the temp.
23 startOfTemp = index  + 1
```

```
24
25  textTemp = text[startOfTemp:endOfTemp]
26  tempInt = int(textTemp)
27  print( "temp is:", textTemp)
```

As you can see, we spend three times as much code finding the temperature within the HTML as we do actually downloading the webpage!

Using Regular Expressions (RegEx) to Locate Patterns Easily

The re module (short for regular expression) allows us to find specific patterns of text and extract data we want more easily than manually searching for specific characters in the webpage. For a complete introduction to regular expressions, you should read DIVE Chapter 5 . Regular Expressions are very powerful and can be used to solve many parsing problems.

In this example we will demonstrate how to use regular expressions to extract the piece of text we want from the AJC's webpage. Specifically, we are looking for "NN°" where the NN is replaced with digits. Note that on extremely hot days it could be NNN and on extremely cold days it could be just N. (We will assume that Atlanta, Georgia will never have a -N temperature day.)

We will use a two step process to get the information we want. First, we will use a regular expression to match and extract the text around the temperature. After we have this text (which contains only one number) we will extract just the number.

Here is the code:

```
1   import urllib.request
2   from re import findall
3
4   #Read the webpage:
5   response = urllib.request.urlopen("http://www.ajc.com")
6   html = response.read()
7   text = html.decode()
8
9   #Use regular expressions to find the data we want, which looks like:
10  #   "<span>NN&deg;</span>" where the NN is replaced with digits.
11  # Note that on extremely hot days it could be NNN and on extremely
12  # cold days it could be just N.
13
14  dataCrop = findall("<span>[0-9]+&deg;</span>", text)
15  print("The data cropped out of the webpage is:", dataCrop)
16
17  if len(dataCrop) != 1:
18     print("We have a problem parsing the data! Help!")
19  else:
20     #Get just the digits
21     temp = findall("[0-9]+", dataCrop[0])
22     intTemp = int( temp[0] )
23     cTemp = 5/9 * (intTemp-32)
24     print("F temp is:", intTemp)
25     print("C temp is:", cTemp)
```

Note that this version of the code does not use a while loop to walk backwards in the text string one character at a time. Instead, we crop a small chunk of data out of the webpage at line 14. Then (assuming we got only one matching chunk of data) we extract a number from within that data on line 21.

In both cases, we are using the findall method imported from the re module. The findall function will search through a string looking for one or more matches of the regular expression. It will return any matches as a list. In the cases above, we are attempting to match only one specific piece of data. This is why we check to confirm that we have a list of length one at line 17.

The magic of regular expressions allows us to specify that we want one or more digits by using the [0-9] pattern to specify that we will accept any numerical digit, and the plus sign (+) after it to specify that we want one or more digits. Everything else in the regular expression must match exactly, which means that we must have a tag in front of our digits and a HTML degree symbol (°) followed by an tag.

The second regular expression on line 21 is simply the part that only matches numerical digits, and is used to extract just the numbers. Note that if we wanted to accept negative temperatures we would need to use a regular expression that allowed an optional (zero or one) dash in front of the number, such as the following: "-?[0-9]+". The question mark character after a pattern indicates that we are looking for zero or one repetitions of the patter (a dash in this example).

Here is a table of regular expression patterns you should know:

Pattern	What it does
.	Matches any character (including spaces!) (excluding newlines if dotall is not set to True)
\d	Matches all digits
\D	Matches non-digits
\s	Matches any whitespace character (newline, tab, space, etc)
\S	Matches any non-whitespace character
[abc]	Matches the character a, b, or c
[a-zA-Z]	Matches any character in the range a-z or A-Z
[^<]	Matches any character OTHER than the < bracket.
p+	Matches 1 or more instances of pattern p
p?	Matches 0 or 1 instances of pattern p
p*	Matches 0 or more instances of pattern p
p{m}	Specific Number of patterns: Matches exactly m instances of pattern p
p{m,n}	Specific range of numbers of a pattern: Matches m to n instances of pattern p
|	A|B Matches either A or B
(p)	Capturing Group - Groups multiple patterns together such that everything inside the group is considered one pattern. Only the data within a capturing group (or groups) will be returned by the findall function call.
(?:p)	Non-Capturing Group - Like a capturing group, but doesn't cause python to split up what's in the group from the rest of the pattern when you use findall. This is useful if you need to repeat the entire pattern inside the group multiple times, but want to capture the entire RegEx.

Note that some characters have special meanings within a regular expression, So if you want to match one of those characters you will probably have to "escape" it much like you escape a newline or tab character, using a backslash. Here are some examples of how you can match these "special" characters.

Pattern	What it does
.	Matches any character (including spaces!) (excluding newlines if dotall is not set to True)
\.	Matches (only) a period in the text.
$	Matches the end of a string (NOT a dollar sign!)
\$	Matches a dollar sign in the text
\(	Matches a begin parenthesis that is in the text, and is not intended to be the beginning of a capturing group.

Regex Example: Extracting addresses

Consider this Table of sample Address data. You could write code using while loops and if statements to extract the address data, but it would be easier to extract using Regular Expressions.

The following code downloads the website and extracts all of the phone numbers:

```
1  import urllib.request
2  from re import findall
3  url = "http://www.summet.com/dmsi/html/codesamples/addresses.html"
4  response = urllib.request.urlopen(url)
5  html = response.read()
6  htmlStr = html.decode()
7
8  pdata = findall("\(\d{3}\) \d{3}-\d{4}", htmlStr)
9  for item in pdata:
10    print(item)
```

Note the use of the regular expression on line 8. It makes use of the “digits” pattern \d along with the “specific number” modifier {N}. It also has to use backslash characters to escape the parenthesis so that they match the parenthesis in the phone number text, and do not act as a capturing group.

If you wanted to capture the First and Last names the Regular Expression gets more complicated. You can’t simply grab any two words, as addresses have sets of two words in them. Instead, you need to anchor the RegEx match to the front of the record. One way to do this is by examining the underlying HTML code and noting that each line starts with an <li> tag and looks like the following:

```
<li>Cecilia Chapman<br/>711-2880 Nulla St.<br/>Mankato Mississippi 96522<br/>(257) 563-7401</li>
```

We can use the starting list item tag <li> to signal the beginning of a record, and then grab the next two words that are separated by a space using the following Regular Expression:

```
<li>([A-Za-z]+ [A-Za-z]+)<br/>
```

Note that we use a single capturing group (in parenthesis) to indicate that we want to extract the Name data, but NOT the <li> and
 tags that bracket it. Those tags are still part of the regular expression, and are used for matching, but are not returned as part of the data that is gathered when we run the regular expression in the findall method:

```
ndata = findall("<li>([A-Za-z]+ [A-Za-z]+)<br/>", htmlStr)
for item in ndata:
  print(item)
```

Because we only ask to capture a single group, the data is returned as a list of strings, where each string is a full name containing both the first and last name. If we had wanted to capture the first and last name separately, we could have used code such as the following:

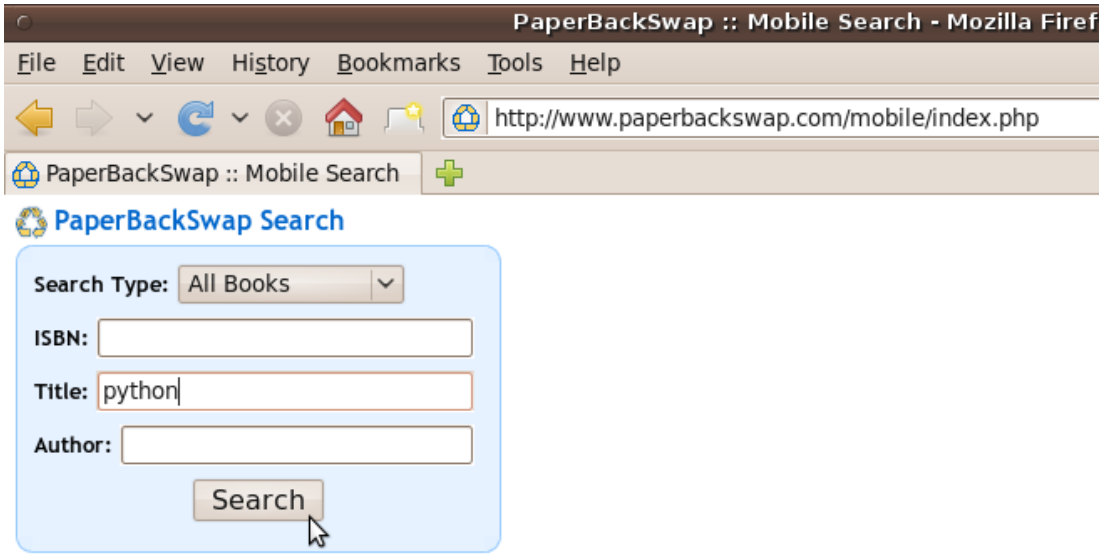
```
ndata = findall("<li>([A-Za-z]+) ([A-Za-z]+)<br/>", htmlStr)
for item in ndata:
    print(item)
```

The only difference is that it has two capturing groups (patterns in parenthesis), so the data will be returned as a list of tuples, and each tuple will contain two strings, the first and last name.

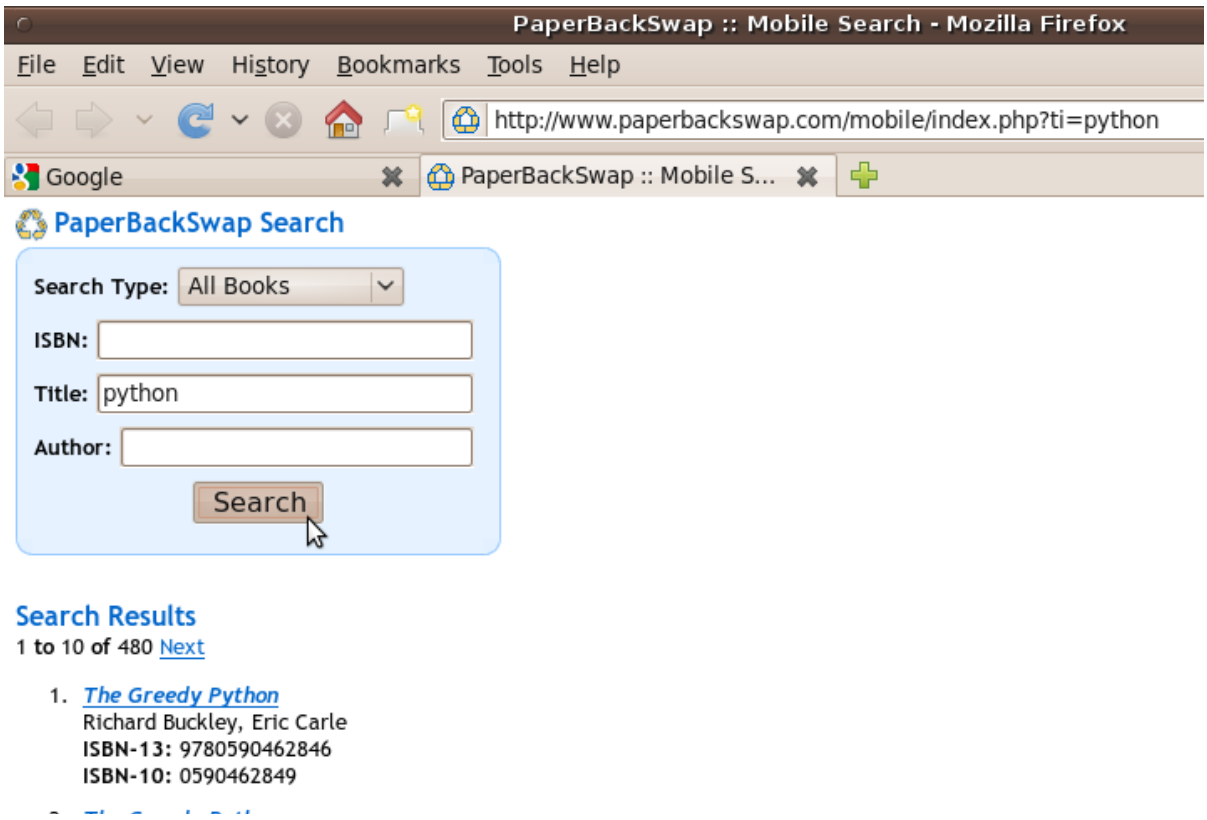
Requesting Dynamic webpages: A Simple GET Request

Many websites ask you to fill out a form, search box, or drop-down list to select the information you are interested in. For example, stock market websites typically ask you to type the stock symbol that you would like to look up, and weather websites ask for your city and state or zip code to provide personalized weather reports.

Much of this data is encoded as a “GET” request in the URL of the request. For example, if you wanted to search for books that had the world “Python” in the title on PaperBackSwap.com’s mobile website, you would type “Python” in the Title search field and press the search button.



When you do this, your web browser automatically encodes the data you have entered and attaches it to the URL, generating a URL that includes "&ti=python" at the end. (The ti stands for Title, and python is the keyword you searched for.) The page returned has the same form on the top, and includes some search results below.



Of course, you don't actually have to load the webpage in a browser, type "python" in the title field, and press the search button to get the page. Instead, you can simply request the appropriate URL:

http://www.paperbackswap.com/mobile/index.php?ti=python

using code such as the following:

```
1  import urllib.request
2
3  request = urllib.request.Request("http://www.paperbackswap.com/mobile/index.php?ti=python")
4
```

```
response = urllib.request.urlopen(request)
the_page = response.read()
theText = the_page.decode()
```

In general, any request made using a HTTP GET can be seen in the URL, and you can simply copy the appropriate URL form a browser and request it again and again to get the same data for your program. By convention, GET requests do not change “state” on the server, so they are typically safe to make.

Filling out POST Forms

Sometimes you need to fill out a form to request certain information, and the form does NOT use HTTP GET to send it’s request to the server. A form that uses HTTP POST will not encode the user data in the URL, and will instead send the user data as part of the request headers. For an example of this type of form, see the following URL:

http://www.tizag.com/phpT/examples/formexample.php

When you fill this form out, it goes to the server and gets a page (which simply echos the data you entered), but it does not encode the data you entered as part of the URL. If you wanted to download the page automatically, you would need to use the urllib.parse module to embed the proper data into the headers of the request. Here is an example that puts my name in the firstname and lastname fields, selects “Steak” as my favorite food from the checkbox options. Note that the names of the input fields is embedded in the HTML of the form page, and you may need to View->Page Source in a web browser to determine the correct names and options for a POST form.

Also note that this php page both generates the html for the form, and (if the “submit” field is passed back with a value of “submit”) processes the form once the submit button is pressed.

```
import urllib.parse
import urllib.request

#Form URL: http://www.tizag.com/phpT/examples/formexample.php



#URL of CGI script it calls:
url = "http://www.tizag.com/phpT/examples/formexample.php"

values = { 'Fname' : "Jay",
           'Lname' : "Summet",
            'food[]': "Steak",
            "TofD": "Day",
            "submit": "submit"  #This PHP example checks for this entry!
           }

data = urllib.parse.urlencode(values)
data = data.encode('ascii')   #Needed for Python 3.2 and above!
req = urllib.request.Request(url,data)

response = urllib.request.urlopen(req)
the_page = response.read()  #As bytes!
#print(the_page)
pageStr = the_page.decode()         #As a string!

#Find the printed data...
helloAt = pageStr.find("Hello")
print("Hello at:", helloAt )
print(pageStr[helloAt:helloAt+300] )
```

Typically, POST forms are used to change state on the server. (For example, to update your email address, etc) Because of this, you should take a bit of extra care when POSTing data to a web server. However, this can sometimes be useful if you want to write a program that takes some action for the user and updates a website

dynamically.

Parsing HTML Tables

Once you have the HTML of the website, you must parse it to extract the data. Much of the data on the web is presented using HTML tables, such as this example: http://www.summet.com/dmsi/html/codesamples/simpleTable.html

Python does have modules which can be used to parse HTML, but sometimes it is easier to write your own code for simple table parsing.

The code example below will go through a string of text (HTML from a sample webpage that has an HTML table) looking for tags. When it finds a begin table tag (<table>) it will remember that it is inside a table by setting the inTable boolean to True (line 37). When it finds a begin table row tag (<tr>), it will start a new ourTR variable to hold the TD data items, and set the inTR variable to True (lines 42-44). When it finds a start table data tag (<td>) it will start a new ourTD variable, and set the inTD variable to true (lines 54-56).

When the while loop knows that it is inside a table, inside a row, and inside a TD item (line 81), it will copy the text (not including tags) to the ourTD variable (line 82) until it hits an end table data tag (</td>) (line 59) at which point it will append that table data item (in the ourTD variable) to the ourTR list (lines60-64). When it hits the end of a table row tag (</tr>) it will append the ourTR list of table data items into the ourTable list (lines 49-51). Once it hits the end of the HTML or runs out of tags, it will return the ourTable data.

(Download readTable.py)

```
1  import urllib.request
2
3  def parseTable(html):
4      #Each "row" of the HTML table will be a list, and the items
5      #in that list will be the TD data items.
6      ourTable = []
7
8      #We keep these set to NONE when not actively building a
9      #row of data or a data item.
10     ourTD = None    #Stores one table data item
11     ourTR = None    #List to store each of the TD items in.
12
13
14     #State we keep track of
15     inTable = False
16     inTR = False
17     inTD = False
18
19     #Start looking for a start tag at the beginning!
20     tagStart = html.find("<", 0)
21
22     while( tagStart != -1):
23         tagEnd = html.find(">", tagStart)
24
25         if tagEnd == -1:    #We are done, return the data!
26             return ourTable
27
28         tagText = html[tagStart+1:tagEnd]
29
30         #only look at the text immediately following the <
31         tagList = tagText.split()
32         tag = tagList[0]
33         tag = tag.lower()
34
35         #Watch out for TABLE (start/stop) tags!
36         if tag == "table":      #We entered the table!
37             inTable = True
38         if tag == "/table":     #We exited a table.
39             inTable = False
```

```
40
41          #Detect/Handle Table Rows (TR's)
42          if tag == "tr":
43              inTR = True
44              ourTR = []      #Started a new Table Row!
45
46          #If we are at the end of a row, add the data we collected
47          #so far to the main list of table data.
48          if tag == "/tr":
49              inTR = False
50              ourTable.append(ourTR)
51              ourTR = None
52
53          #We are starting a Data item!
54          if tag== "td":
55              inTD = True
56              ourTD = ""      #Start with an empty item!
57
58          #We are ending a data item!
59          if tag == "/td":
60              inTD = False
61              if ourTD != None and ourTR != None:
62                  cleanedTD = ourTD.strip()   #Remove extra spaces
63                  ourTR.append( ourTD.strip() )
64              ourTD = None
65
66
67          #Look for the NEXT start tag. Anything between the current
68          #end tag and the next Start Tag is potential data!
69          tagStart = html.find("<", tagEnd+1)
70
71          #If we are in a Table, and in a Row and also in a TD,
72          # Save anything that's not a tag! (between tags)
73          #
74          #Note that this may happen multiple times if the table
75          #data has tags inside of it!
76          #e.g. <td>some <b>bold</b> text</td>
77          #
78          #Because of this, we need to be sure to put a space between each
79          #item that may have tags separating them. We remove any extra
80          #spaces (above) before we append the ourTD data to the ourTR list.
81          if inTable and inTR and inTD:
82              ourTD = ourTD + html[tagEnd+1:tagStart] + " "
83              #print("td:", ourTD)   #for debugging
84
85
86      #If we end the while loop looking for the next start tag, we
87      #are done, return ourTable of data.
88      return(ourTable)
89
90
91
92
93
94
95  response = urllib.request.urlopen("http://www.summet.com/dmsi/html/codesamples/simpleTable.html")
96  html_bytes = response.read()
97  html = html_bytes.decode()
98  print(html)
99
100 dataTable = parseTable(html)
101 print(dataTable)
```