<https://cambridgespark.com/collecting-web-data-without-an-api/>

In this tutorial, we’ll show you how to extract data from Wikipedia pages.

If you’ve ever gone through an online machine learning tutorial, you’re likely to be familiar with standard datasets like [Titanic casualties](https://github.com/mwaskom/seaborn-data/blob/master/titanic.csv), [Iris flowers](https://github.com/mwaskom/seaborn-data/blob/master/iris.csv) or [customer tips](https://github.com/mwaskom/seaborn-data/blob/master/tips.csv). These simple, well-structured datasets are great for getting to grips with data science fundamentals. And once you’ve mastered them, you can create your own datasets to investigate anything at all. You might use public APIs to gather this data, such as the ones available for [Twitter](https://developer.twitter.com/en/docs.html), [Reddit](https://www.reddit.com/dev/api/" \t "_blank)or [Instagram](https://www.instagram.com/developer/). Many APIs also have Python packages which make it even easier to access the data you want: [Tweepy](http://www.tweepy.org/" \t "_blank) for Twitter or [PRAW](https://praw.readthedocs.io/en/latest/) for Reddit.

What about when you want to gather data from a website that doesn’t have a nice convenient API? If there’s not much data, it might be possible to just manually copy and paste it — a bit tedious, perhaps, but not too much trouble. And if you just want to extract a table or two from a webpage, you can even use [pandas.read\_html()](https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_html.html" \t "_blank) to load and parse the page, automatically extracting tables as dataframes.

**Crash course: HTML tables**

If you are not familiar with HTML, take some time to read about [HTML element tags](https://www.w3schools.com/tags/default.asp), [HTML element attributes](https://www.w3schools.com/html/html_attributes.asp) and [HTML classes](https://www.w3schools.com/html/html_classes.asp). Knowing about these parts of HTML will make it much easier for us to identify and extract the data we want.

Tables in HTML have a nested structure. For a fuller introduction, take a look at the [w3schools.com](https://www.w3schools.com/html/html_tables.asp) guide. Key points are that nested inside a tableelement are tr elements, which represent the rows. And nested inside the rows are th or td elements, representing the actual cells. th denotes the header row of the table, while td elements will be the standard cells. In the following code, you can see how some elements have class or hrefattributes.

1. <table class="main-table">
2. <tr>
3. <th>Letter</th>
4. <th>Number</th>
5. <th>Country</th>
6. <th>Link</th>
7. </tr>
8. <tr>
9. <td>A</td>
10. <td>1</td>
11. <td>Germany</td>
12. <td><a href="www.url.com">Germany</a>
13. </tr>
14. <tr>
15. <td>B</td>
16. <td>2</td>
17. <td>France</td>
18. <td><a href="www.url.com">France</a>
19. </tr>
20. </table>

**Which generates this table:**

Tables can be much more complex than this, as we will see, but this nested structure is common to them all.

Extracting tables from HTML is easy with pandas:

**In [1]:**

1. import pandas as pd
2. data = pd.**read\_html**('https://en.wikipedia.org/wiki/List\_of\_cities\_in\_the\_United\_Kingdom')
3. **print**(f'Extracted {len(data)} table/s')
4. data[0]
5. Extracted 4 table/s

**Out[1]:**

If we look at the first item extracted, it’s the table showing information about all the cities in the UK. However, this would need cleaned up a bit — there are no column names, plus there is a lot of extraneous text such as footnote references and so on. This data is not very useable in its current format.

**Crawling**

Imagine you want to not only get a list of all the cities in the UK, but extract some information about them from Wikipedia. And you don’t want to click on anything! We’ll use [requests](http://docs.python-requests.org/en/master/) and [beautifulsoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/" \t "_blank) to do this. Wikipedia articles like these contain a vast amount of information, but we will focus on the following:

* Which country it is in
* The current population
* A list of other Wikipedia pages the city article links to in its introduction

The first three items on this list will require extracting data from tables. The last item will extract data that is not in a table, which pandas would not be able to extract.

The process looks like this:

1. Generate a list of Wikipedia URLs for the cities of the UK
2. Access each URL and extract the above information
3. Save the extracted information to disk for analysis later on

**Extracting the URLs from a Wikipedia page**

Although we can use pandas to get the text of the table of cities, it doesn’t get the URLs. Another option would be to just copy and paste the links from the page, but that almost defeats the point of knowing Python. We can automate this process.

There are some packages that can do this for us if we have some knowledge about how HTML is structured:

* [requests](http://docs.python-requests.org/en/master/) to get the HTML. This is a great package for accessing URLs and is incredibly simple to use
* [beautifulsoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/) to parse that HTML and extract the items we want. Also very simple to use, but it helps to know a bit about HTML beforehand

**In [2]:**

1. import requests
2. city\_list = requests.**get**('https://en.wikipedia.org/wiki/List\_of\_cities\_in\_the\_United\_Kingdom')
3. **print**(city\_list.status\_code, city\_list.reason, **len**(city\_list.text))
4. **print**(city\_list.text[0:500])
5. 200 OK 301965
6. <!DOCTYPE html>
7. <html class="client-nojs" lang="en" dir="ltr">
8. <head>
9. <meta charset="UTF-8"/>
10. <title>List of cities in the United Kingdom - Wikipedia</title>
11. <script>document.documentElement.className = document.documentElement.className.**replace**( /(^|\s)client-**nojs**(\s|$)/, "$1client-js$2" );</script>
12. <script>(window.RLQ=window.RLQ||[]).**push**(**function**(){mw.config.**set**({"wgCanonicalNamespace":"","wgCanonicalSpecialPageName":false,"wgNamespaceNumber":0,"wgPageName":"List\_of\_cities\_in\_the\_United\_Kingd

The city\_list variable contains all the HTTP data for the URL we accessed. The request was successful, as a [status code](https://en.wikipedia.org/wiki/List_of_HTTP_status_codes) of 200 was returned, and around 300k characters of HTML were retrieved.

Before information can be extracted from that HTML, it must be parsed. This lets us refer to elements in the data according to various properties (such as class or attribute) and is much easier than using regular expressions.

At this point, we need to find the “List of cities” table in the HTML. This can be done by viewing the page source (available from the right-click menu in most web browsers) and searching for <table to find the start of any tables. This page contains four but the one we want looks like this:

This table is of the class wikitable sortable. We can use [beautifulsoup](https://www.crummy.com/software/BeautifulSoup/bs4/doc/" \t "_blank) to parse the HTML and search for all table elements with a class of wikitable sortable.

**In [3]:**

1. from bs4 import BeautifulSoup as bs
2. ​city\_list = requests.**get**('https://en.wikipedia.org/wiki/List\_of\_cities\_in\_the\_United\_Kingdom')
3. city\_list = **bs**(city\_list.text, "lxml")
4. table = city\_list.**find\_all**('table', {'class':'wikitable sortable'})
5. **print**(f'Found {len(table)} table/tables.')
6. table = table[0]
7. Found 1 table/tables.

Luckily, there is only one table with that class, so there is no need to examine all the results and find the correct item. We can just use the single item in the list that the find\_all() function returns.

HTML tables are organised by rows, which are tr elements inside the parent table. These can be extracted using find\_all() again.

**In [4]:**

1. table\_rows = table.**find\_all**('tr')
2. **print**(f'Found {len(table\_rows)} row/rows.')
3. Found 70 row/rows.

There are only 69 cities in the UK, so it looks like we’ve got the data we need plus an extra header row.

The cells in each row are td elements. The exception is the header row, which are th. Again, these can be extracted from each row using find\_all(). We only want the first column of the table, which has the links to the cities’ pages. This link is an a element, which can be extracted just like the other elements. It’s clear from the HTML source that the first link is what we want. Once this has been found, it is the href attribute that needs to be extracted – this is the actual link. Finally, the links will be relative rather than absolute, so it will be necessary to prepend the domain name onto them.

**In [5]:**

1. links = []
2. for row in table\_rows:
3. cells = row.**find\_all**('td')
4. if cells:
5. link = cells[0].**find\_all**('a')[0]
6. link = link.attrs['href']
7. links.**append**(f'https://en.wikipedia.org{link}')
8. else:
9. # We found "th" header cells, instead of "td"
10. pass
11. **print**(f'Extracted {len(links)} links. Is this the right number? {"Yes" if len(links) == 69 else "No"}.')
12. links[0:4]
13. Extracted 69 links. Is this the right number? Yes.

**Out[5]:**

1. ['https://en.wikipedia.org/wiki/Aberdeen',
2. 'https://en.wikipedia.org/wiki/Armagh',
3. 'https://en.wikipedia.org/wiki/Bangor,\_Gwynedd',
4. 'https://en.wikipedia.org/wiki/Bath,\_Somerset']

Now that we have our links, we can loop through them and extract required information from each page. In the following part, I’ll use the example of [Leeds](https://en.wikipedia.org/wiki/City_of_Leeds) to show you how to use the structure of HTML to extract what we need.

**Which country is the city in?**

This information can be found in the info box to the right of the Wikipedia page, next to the cell named “Constituent country”.

Inspecting the HTML source shows that this is inside a tr of class mergedrow. There is a header th cell with the text “Constituent country” and the data we want is in the text of the adjacent td cell.

(**Note**: When trying to examine the structure of the HTML, you will probably find it useful to view it in an editor that does syntax highlighting and will prettify the code to show the nested structure. Many text editors, such as [SublimeText](https://www.sublimetext.com/3" \t "_blank), have plugins that will do this if they do not do it natively.)

To extract this, we can search for all the mergedrow elements. Then we loop through all the pairs until we find what we want.

**In [6]:**

1. city = requests.**get**('https://en.wikipedia.org/wiki/City\_of\_Leeds')
2. city = **bs**(city.text, 'lxml')
3. ​rows = city.**find\_all**('tr', {'class':'mergedrow'})
4. ​for header, cell in rows:
5. if 'country' in header.text:
6. **print**(f'Country is {cell.text.strip()}')
7. # Use .strip() because the text has linebreaks in it that we want to remove
8. Country is England

**Testing**

This works for Leeds, but what about the other cities? Unfortunately, the Wikipedia pages are not entirely consistent in how they structure their HTML. It will be necessary to check the output.

**In [7]:**

1. country\_data = {}
2. for c in links:
3. name = c.**split**('/')[-1]
4. # Get the city name from the URL
5. city = requests.**get**(c)
6. city = **bs**(city.text, 'lxml')
7. rows = city.**find\_all**('tr', {'class':'mergedrow'})
8. ​for r in rows:
9. if **len**(r) == 2:
10. header, cell = r
11. if 'country' in header.text.**lower**():
12. country\_data[name] = cell.text.**strip**()
13. # No need to keep looking once a match is found
14. break
15. if country\_data.**get**(name):
16. # Evaluates as True only if key in dictionary
17. pass
18. else:
19. **print**(f'Did not get data for {name}!')
20. **print**(f'Got data for {len(country\_data)} of {len(links)} cities.')
21. # If a country was found for the city, there will be a dictionary entry for it.
22. Did not get data for Newport,\_Wales!
23. Got data for 68 of 69 cities.

So one city is missing the data we need. That’s easily checked and it seems the problem is that the Wikipedia entry for the city of Newport in Wales uses the word “part” instead of “country”

(Note: this is a very common problem when scraping data from HTML! You think you’ve figured it all out and then the edge cases start popping up, where some tiny difference makes it impossible to extract what you want.)

This is easily fixed by checking for “part” and “country”:

**In [8]:**

1. country\_data = {}
2. for c in links:
3. name = c.**split**('/')[-1]
4. # Get the city name from the URL
5. city = requests.**get**(c)
6. city = **bs**(city.text, 'lxml')
7. rows = city.**find\_all**('tr', {'class':'mergedrow'})
8. for r in rows:
9. if **len**(r) == 2:
10. header, cell = r
11. if 'country' in header.text.**lower**() or 'part' in header.text.**lower**():
12. country\_data[name] = cell.text.**strip**()
13. # No need to keep looking once a match is found
14. break
15. if country\_data.**get**(name):
16. # Evaluates as True only if key in dictionary
17. pass
18. else:
19. **print**(f'Did not get data for {name}!')
20. **print**(f'Got data for {len(country\_data)} of {len(links)} cities.')
21. # If a country was found for the city, there will be a dictionary entry for it.
22. Got data for 69 of 69 cities.

**What is the current population?**

The HTML source shows that this is going to be trickier: the population data is not neatly marked with a convenient header in the table. From looking at the entry for Leeds:

The data we want is inside a td element but this has no useful attributes we can use to identify it. However, the preceding tr element, with class mergedtoprow, contains the word “Population”. So we can target that and get its next sibling in the HTML structure. This is the tr with mergedrow: then we just need to get its children and select the td element, which we know (from the HTML) is the second child in the returned iterator. The text can be extracted from this and cleaned up.

This time, let’s check all the cities at once and clean up as we go.

**In [9]:**

1. population\_data = {}
2. for c in links:
3. name = c.**split**('/')[-1]
4. # Get the city name from the URL
5. city = requests.**get**(c)
6. city = **bs**(city.text, 'lxml')
7. rows = city.**find\_all**('tr', {'class':'mergedtoprow'})
8. population = None
9. for r in rows:
10. if 'Population' in r.text:
11. sibling = r.next\_sibling
12. td\_child = **list**(sibling.children)[1]
13. population = td\_child.text.**strip**()
14. # Remove linebreaks from start/end
15. population = population.**split**()[0]
16. # Get the first item before a space
17. population\_data[name] = population
18. break
19. **print**(name, population)
20. Aberdeen 3,505.7/km2
21. Armagh None
22. Bangor,\_Gwynedd None
23. Bath,\_Somerset None
24. Belfast None
25. Birmingham 1,137,100
26. City\_of\_Bradford 534,800
27. Brighton\_and\_Hove 288,200
28. Bristol 459,300
29. Cambridge 124,900
30. City\_of\_Canterbury 164,100
31. Cardiff 361,468[1]
32. City\_of\_Carlisle 108,300
33. City\_of\_Chelmsford 176,200
34. Chester None
35. Chichester None
36. Coventry 360,100
37. Derby 248,700
38. Derry None
39. Dundee 148,270
40. Durham,\_England 48,069
41. Edinburgh 464,990
42. Ely,\_Cambridgeshire None
43. Exeter 128,900
44. Glasgow 621,020[1]
45. Gloucester 129,100
46. Hereford None
47. Inverness None
48. Kingston\_upon\_Hull 260,700
49. City\_of\_Lancaster 142,500
50. City\_of\_Leeds 784,800
51. Leicester 329,839[1]
52. Lichfield None
53. Lincoln,\_England 97,541[1]
54. Lisburn None
55. Liverpool 491,500
56. City\_of\_London 9,401
57. Manchester 545,500
58. Newcastle\_upon\_Tyne 295,800
59. Newport,\_Wales 151,500
60. Newry None
61. Norwich 141,300
62. Nottingham 321,500
63. Oxford 151,906
64. Perth,\_Scotland None
65. City\_of\_Peterborough 198,900
66. Plymouth 263,100
67. Portsmouth 205,100
68. City\_of\_Preston,\_Lancashire 141,300
69. Ripon None
70. **City\_and\_District\_of\_St\_Albans** (of
71. St\_Asaph None
72. St\_Davids None
73. City\_of\_Salford 251,300
74. Salisbury None
75. City\_of\_Sheffield 577,800
76. Southampton 236,900
77. Stirling None
78. Stoke-on-Trent 255,400
79. City\_of\_Sunderland 277,962
80. Swansea Unitary
81. Truro None
82. City\_of\_Wakefield 340,800
83. Wells,\_Somerset None
84. City\_of\_Westminster 244,800
85. City\_of\_Winchester 123,900
86. Wolverhampton 259,900
87. Worcester 101,328
88. City\_of\_York 208,200

Oh dear! It looks like we’ve grabbed the wrong HTML element for some cities and instead got the area, or failed to get anything at all.

To fix this, we need to dig around in the HTML for cities that failed and edit our code. Here’s what the result of that should look like.

**In [10]:**

1. population\_data = {}
2. for c in links:
3. name = c.**split**('/')[-1]
4. # Get the city name from the URL
5. city = requests.**get**(c)
6. city = **bs**(city.text, 'lxml')
7. population = None
8. if name in ['Derry']:
9. rows = city.**find\_all**('tr')
10. for r in rows:
11. if 'Population' in r.text:
12. population = r.**find\_all**('li')[0].text
13. break
14. elif name in ['Aberdeen', 'Armagh', 'Bangor,\_Gwynedd', 'Bath,\_Somerset', 'Belfast', 'Chester','Chichester', 'Hereford', 'Inverness', 'Ely,\_Cambridgeshire', 'Lichfield', 'Lisburn', 'Newry', 'Perth,\_Scotland', 'Ripon', 'Stirling', 'Truro', 'Wells,\_Somerset']:
15. rows = city.**find\_all**('tr')
16. for r in rows:
17. if 'Population' in r.text:
18. children = **list**(r.children)
19. population = children[1].text.**replace**('\n', ' ').**strip**()
20. break
21. else:
22. rows = city.**find\_all**('tr', {'class':'mergedtoprow'})
23. for r in rows:
24. if 'Population ' in r.text:
25. population = r.next\_sibling.text.**replace**('\n', ' ').**strip**()
26. break
27. population\_data[name] = population

**In [11]:**

1. for name, population in population\_data.**items**():
2. if population:
3. **print**(f'{name:<30} : {population:}')
4. else:
5. **print**(f'!! Nothing found for {name}')
6. Aberdeen : 196,670[1] – City proper
7. Armagh : 14,**749** (2011 Census)
8. Bangor,\_Gwynedd : 18,**810** (2011 census)
9. Bath,\_Somerset : 88,859 [1]
10. Belfast : City of Belfast:340,**200** (2017)[2] Urban Area:483,**418** (2016)[3]Metropolitan area:671,**559** (2011)[4]
11. Birmingham : • City 1,137,100
12. City\_of\_Bradford : • Total 534,**800** (Ranked 6th)
13. Brighton\_and\_Hove : • City and unitary authority 288,200
14. Bristol : • City and county 459,**300** (Ranked 10th district and 43rd ceremonial county)
15. Cambridge : • City and non-metropolitan district 124,**900** (ranked 183rd)
16. City\_of\_Canterbury : • Total 164,100
17. Cardiff : • City & County 361,468[1]
18. City\_of\_Carlisle : • Total 108,**300** (Ranked 221st)
19. City\_of\_Chelmsford : • Total 176,200
20. Chester : 118,200 [1]
21. Chichester : 26,795 [2] 2011 Census
22. Coventry : • City and Metropolitan borough 360,**100** (Ranked 15th)
23. Derby : • City and Unitary authority area 248,700
24. Derry : Derry: 85,016
25. Dundee : • Total 148,270
26. Durham,\_England : • Total 48,**069** (urban area)[1]
27. Edinburgh : • City and council area 464,990 – Locality [1] 513,210 – Local Authority Area[2]
28. Ely,\_Cambridgeshire : 20,112
29. Exeter : • Total 128,900
30. Glasgow : • City 621,020[1]
31. Gloucester : • Total 129,100
32. Hereford : 58,896 [1]
33. Inverness : 63,780 [1]
34. Kingston\_upon\_Hull : • City 260,**700** (Ranked 58th)
35. City\_of\_Lancaster : • Total 142,500
36. City\_of\_Leeds : • Total 784,**800** (Ranked 2nd)
37. Leicester : • City 329,839[1]
38. Lichfield : 32,**219** (2011)[2]
39. Lincoln,\_England : • City and Borough 97,541[1]
40. Lisburn : 120,465 surrounding areas
41. Liverpool : • City 491,500
42. City\_of\_London : • City 9,401
43. Manchester : • City 545,500
44. Newcastle\_upon\_Tyne : • City 295,**800** (ranked 40th district)
45. Newport,\_Wales : • Total 151,**500** (Ranked 7th)
46. Newry : 26,**967** (2011)[4]
47. Norwich : • City 141,**300** (ranked 146th)
48. Nottingham : • City 321,500
49. Oxford : • City and non-metropolitan district 151,**906** (2,011)[1]
50. Perth,\_Scotland : 47,**180** (est. 2012), excluding suburbs[3]
51. City\_of\_Peterborough : • Total 198,900
52. Plymouth : • Total 263,100
53. Portsmouth : • City & unitary authority area 205,**100** (Ranked 76th)[2]
54. City\_of\_Preston,\_Lancashire : • City & Non-metropolitan district 141,**300** (Ranked 146th)
55. Ripon : 16,**702** (2011 census)[1]
56. City\_and\_District\_of\_St\_Albans : • **Rank** (of 326)
57. !! Nothing found for St\_Asaph
58. !! Nothing found for St\_Davids
59. City\_of\_Salford : • Total 251,**300** (Ranked 65th)
60. !! Nothing found for Salisbury
61. City\_of\_Sheffield : • City 577,**800** (Ranked 3rd)
62. Southampton : • City and unitary authority area 236,**900** (Ranked 57)
63. Stirling : 36,142 Census 2011 [1]
64. Stoke-on-Trent : • City 255,400
65. City\_of\_Sunderland : • Total 277,962
66. Swansea : • Total Unitary Authority area: 245,500, Ranked 2nd Urban area within Unitary Authority: 179,485 Wider Urban Area: 300,352 Metropolitan Area: 462,000 Swansea Bay City Region: 685,051
67. Truro : 18,766 [1]
68. City\_of\_Wakefield : • Total 340,800
69. Wells,\_Somerset : 10,**536** (2011)[1]
70. City\_of\_Westminster : • Total 244,800
71. City\_of\_Winchester : • Total 123,900
72. Wolverhampton : • Total 259,**900** (59th)
73. Worcester : • Total 101,328
74. City\_of\_York : • Total 208,**200** (Ranked 84th)

The structure of the HTML fell into three distinct categories. Derryhad a list structure inside a table cell. Several cities had rows without the mergedtoprowclass name. But most used the same structure as Leeds.

Some cities did not actually have population data listed inside tables (e.g. St.Asaph, St.David). This will likely have to be added manually.

A final issue is that the data extracted is still very messy. Ideally it would be nicely formatted as integers. However, a few [regular expressions](https://docs.python.org/3/library/re.html) will fix that all up.

**Extracting links**

The final bit of information we’ll extract is all the links contained in the introduction for each city. In general, the introduction is the text that appears before the table of contents. Each paragraph is wrapped in a p element in HTML and the table of contents has an id of toc. Because id [must be unique](https://www.w3schools.com/html/html_id.asp) in an HTML document, it will be easy to identify the TOC. The Wiki pages we are working with here have the general structure:

First comes the infobox table, which we’ve already extracted data from. Then a series of paragraphs of text. Then a div containing the table of contents. All we need to do is loop through the p elements and keep checking to see if that element has a sibling below it on the page which has a particular id and, if so, stop looping through the paragraphs. And for each paragraph, we’ll extract the a elements to get the links.

**In [12]:**

1. intro\_data = {}
2. for c in links:
3. name = c.**split**('/')[-1]
4. # Get the city name from the URL
5. city = requests.**get**(c)
6. city = **bs**(city.text, 'lxml')
7. intro\_links = []
8. introduction\_paragraphs = []
9. all\_paragraphs = city.**find\_all**('p')
10. for p in all\_paragraphs:
11. # Find all the links inside a paragraph
12. found\_links = **set**([l.attrs['href'] for l in p.**find\_all**('a')])
13. intro\_links.**extend**([i for i in found\_links if i.**startswith**('/wiki/') and ':' not in i])
14. # Only want links that go to Wikipedia pages, but exclude Help: pages
15. next\_sibling = p.**find\_next\_sibling**()
16. # Check the next item in the HTML structure that comes after this paragraph
17. if next\_sibling:
18. # If it exists...
19. if next\_sibling.**get**('id') == 'toc':
20. # And if it has the toc id
21. break
22. # Then have eached TOC, so stop processing further paragraphs.
23. intro\_data[name] = intro\_links

**In [13]:**

1. for city, link\_list in **list**(intro\_data.**items**())[0:10]:
2. **print**(f'{len(link\_list):>3} links for {city}')
3. **print**(intro\_data['Aberdeen'][0:5])
4. 23 links for Aberdeen
5. 23 links for Armagh
6. 12 links for Bangor,\_Gwynedd
7. 41 links for Bath,\_Somerset
8. 20 links for Belfast
9. 47 links for Birmingham
10. 53 links for City\_of\_Bradford
11. 8 links for Brighton\_and\_Hove
12. 58 links for Bristol
13. 32 links for Cambridge
14. ['/wiki/List\_of\_towns\_and\_cities\_in\_Scotland\_by\_population', '/wiki/City\_status\_in\_the\_United\_Kingdom', '/wiki/List\_of\_urban\_areas\_in\_the\_United\_Kingdom', '/wiki/Local\_government\_in\_Scotland', '/wiki/Scots\_language']

**Putting it all together**

With the code written for extracting the three required kinds of information, we can put together a little script that will do everything.

**In [14]:**

1. import requests
2. from bs4 import BeautifulSoup as bs
3. links = [] # Store links to city Wiki pages
4. city\_list = requests.**get**('https://en.wikipedia.org/wiki/List\_of\_cities\_in\_the\_United\_Kingdom')
5. city\_list = **bs**(city\_list.text, "lxml")
6. table = city\_list.**find\_all**('table', {'class':'wikitable sortable'})
7. table = table[0]
8. # Download and parse HTML list of UK cities
9. for row in table\_rows:
10. cells = row.**find\_all**('td')
11. if cells:
12. link = cells[0].**find\_all**('a')[0]
13. link = link.attrs['href']
14. links.**append**(f'https://en.wikipedia.org{link}')
15. else:
16. # We found "th" header cells, instead of "td"
17. pass
18. country\_data = {} # Store info about which country each city is in
19. population\_data = {} # Store info about the population of each city
20. intro\_data = {} # Store info about links in city introduction text
21. for c in links:
22. name = c.**split**('/')[-1]
23. # Get the city name from the URL
24. city = requests.**get**(c)
25. city = **bs**(city.text, 'lxml')
26. # Download and parse the HTML
27. rows = city.**find\_all**('tr', {'class':'mergedrow'})
28. # Access the rows of the table we want
29. for r in rows:
30. if **len**(r) == 2:
31. header, cell = r
32. if 'country' in header.text.**lower**() or 'part' in header.text.**lower**():
33. country\_data[name] = cell.text.**strip**()
34. # No need to keep looking once a match is found
35. break
36. population = None
37. if name in ['Derry']:
38. rows = city.**find\_all**('tr')
39. for r in rows:
40. if 'Population' in r.text:
41. population = r.**find\_all**('li')[0].text
42. break
43. elif name in ['Aberdeen', 'Armagh', 'Bangor,\_Gwynedd', 'Bath,\_Somerset', 'Belfast', 'Chester', 'Chichester', 'Hereford', 'Inverness', 'Ely,\_Cambridgeshire', 'Lichfield', 'Lisburn', 'Newry', 'Perth,\_Scotland', 'Ripon', 'Stirling', 'Truro', 'Wells,\_Somerset']:
44. rows = city.**find\_all**('tr')
45. for r in rows:
46. if 'Population' in r.text:
47. children = **list**(r.children)
48. population = children[1].text.**replace**('\n', ' ').**strip**()
49. break
50. else:
51. rows = city.**find\_all**('tr', {'class':'mergedtoprow'})
52. for r in rows:
53. if 'Population ' in r.text:
54. population = r.next\_sibling.text.**replace**('\n', ' ').**strip**()
55. break
56. population\_data[name] = population
57. intro\_links = []
58. introduction\_paragraphs = []
59. all\_paragraphs = city.**find\_all**('p')
60. for p in all\_paragraphs:
61. # Find all the links inside a paragraph
62. found\_links = **set**([l.attrs['href'] for l in p.**find\_all**('a')])
63. intro\_links.**extend**([i for i in found\_links if i.**startswith**('/wiki/') and ':' not in i])
64. # Only want links that go to Wikipedia pages, but exclude Help: pages
65. next\_sibling = p.**find\_next\_sibling**()
66. # Check the next item in the HTML structure that comes after this paragraph
67. if next\_sibling:
68. # If it exists...
69. if next\_sibling.**get**('id') == 'toc':
70. # And if it has the toc id
71. break
72. # Then have eached TOC, so stop processing further paragraphs.
73. intro\_data[name] = intro\_links

**Collating the data**

The extracted data is in three dictionaries, country\_data, population\_data, intro\_data. There are a variety of ways to structure this for further analysis.

**In [15]:**

1. country\_data\_df = pd.DataFrame.**from\_dict**(country\_data, orient='index', columns=['country'])
2. population\_data\_df = pd.DataFrame.**from\_dict**(population\_data, orient='index', columns=['population'])
3. combined\_data = pd.**concat**([country\_data\_df, population\_data\_df], axis=1)
4. combined\_data.**head**()

**Out[15]:**

**Using the data**

For the link data, it might be interesting to visualise it as a [graph](https://blog.cambridgespark.com/an-introduction-to-graphs-f670d29cc953). That will make it possible to see how cities are connected in terms of the concepts and ideas used to define them in Wikipedia.

First, we’ll get the data into a suitable format for graphing, then save it for use in [Gephi](https://gephi.org/" \t "_blank).

**In [43]:**

1. graph\_data = {'nodes':[], 'links':[]}
2. for city in intro\_data:
3. graph\_data['nodes'].**append**({'name':city, 'group':country\_data\_df.loc[city].country, 'n\_label':city})
4. # Create a node for the city. Get the country it is from our other data, for use in visualisation.
5. # And add a label to it for visualisation too.
6. for link in intro\_data[city]:
7. graph\_data['links'].**append**({'source':city, 'target':link})
8. # Also create a node for each wiki page that is not a city
9. # But no labels for it since we won't visualise those - too messy
10. link\_info = {'name':link, 'group':'Wiki Page', 'n\_label':' '}
11. # Add a link between the city and each wiki page it references.
12. if link\_info not in graph\_data['nodes']:
13. # Need to avoid duplicates
14. graph\_data['nodes'].**append**(link\_info)
15. import igraph
16. g = igraph.Graph.**DictList**(vertices=graph\_data['nodes'], edges=graph\_data['links'], directed=True)
17. g.**write\_graphml**('wikicities.graphml')
18. # Save the data in structure-preserving format for Gephi.

The data can then be loaded into Gephi and visualised. I highly recommend playing around with it! It makes is very easy to arrange the layout of the graph as well as apply colours and labels.

Below, you can see that I’ve coloured the city nodes/labels according to the country they are in. Edges are coloured according to the colour of their source node. It’s clear that cities in the same country share a lot of links in common.

**Next steps**

* Try cleaning up the population data and using it in Gephi to set the size of the nodes for cities, so that larger cities appear bigger in the graph.
* Want to practice working with HTML in beautifulsoup? Try extracting data from [toscrape.com/](http://toscrape.com/), which has lots of different types of webpage to work with.
* If you want to extend the code above to not only follow the links on the [List of UK cities](https://en.wikipedia.org/wiki/List_of_cities_in_the_United_Kingdom) page, but also follow links from *those* links, take a look at the [scrapy](https://scrapy.org/" \t "_blank) framework, which makes it easy to recursively crawl a website.