# Importing Data to a DataFrame

In this chapter we will be covering file formats. Data comes to you in a variety of formats and one of the core tasks of making claims with data is transforming it from one shape or type to another. It’s impossible to cover all file formats or even a fraction of the possible shapes or types. Yet, below are a pretty standard ensemble of file formats. Hopefully, as you become increasingly confident in data wrangling you will see how you can manage other kinds of data having been inspired here.

The first type of data is JSON. It’s the closest to data as it would be stored in Python lists and dictionaries. So it’s not that far a file format from some of the Python basics. The next is HTML and XML. These are certainly different from Python but they should be pretty well structured. Then we move on to CSV, which seems like it should be straightforward but due to the many inconsistencies between implementations, it can be a bit of a grab bag. Next, we will look at Excel. This is actually an XML file in disguise, but we will abstract away from those details by using built-in methods in Python to import it and turn it into a DataFrame. In each case we will use examples of data that you can collect from the web and show how to import it into a DataFrame. However, for this book we have put together permissions for these data so that they can be housed on our GitHub site.

## Before we continue, a word on file organisation

In this book and in the exercises I tend to have the data in a separate file from the code. Now historically, operating systems would use a slash to create the directory structure, such as /user/Bernie/Documents/Book. Unfortunately, Windows and Mac/Linux have different ways of using slash. For \*Nix systems, which includes UNIX, Linux, MacOS, the directory separator is “/” whereas for Windows it is “\”. The way around this in Python is to use the os library (which stands for Operating System) and then use the os.sep variable. This will make sure that it works regardless of operating system. For example: filein = open("..{0}Data{0}test.txt".format(os.sep)) is a platform agnostic way of writing filein = open("../Data/test.txt").

At first, you might think it is simpler to dump all of the data in the folder with the Jupyter notebooks. However, what happens when you are using the same data in multiple notebooks or across multiple chapters? Keeping your data in its own folder helps to keep your code and your analysis organised. I tend to adopt a strategy of having separate folders for: - Code - Data - Output - Writing - Other

In the chapters that follow I will often write my code to read in files from a Data directory that is on the same level as the chapter folders, so wherever we are, you will see “..” (meaning move up a directory) followed by os.sep (meaning slash), Data, slash, and the name of the data set. I encourage you do to the same with your code. You will know it works when the following snippet will run without errors:

import os   
filein = open("..{0}Data{0}test.txt".format(os.sep))  
print(filein.read())

ooops I did it again.

# JSON - JavaScript Object Notation

One of the most popular ways to transfer data on the web is JSON. It stands for JavaScript Object Notation. Although it comes from JavaScript, it looks pretty readable in Python. From a high-level Python perspective, JSON is merely a combination of lists and dictionaries. The following code, which looks like Python, is a dictionary nested in a list nested in a dictionary. Yet, it is actually valid JSON:

muppet\_json = {"Type":"Muppet",  
 "Cases":[{"Kermit":"Frog",  
 "Miss Piggy":"Pig",  
 "Gonzo":"Weirdo"}]}

To load json into Python directly you can use the json library. It provides a means to load data into memory (json.loads(THE\_DATA)) and a means to take a data structure and transform it into valid JSON for writing to disk (json.dumps(THE\_DATA)).

Finding some JSON to work with is not difficult. xx(You can check the website files for an example of json). For example, the website Reddit, which is one of the largest link-sharing communities on the web, will format just about every piece of data on the site as JSON. In order to do this, you simply append .json to the url, as in http://www.reddit.com/r/aww.json. Alternatively, go directly to http://api.reddit.com/r/aww which will also format the data as json by default. IF you go to these URLs on a browser you might end up seeing a page with a complete wall of text. For example:

{"kind": "Listing", "data": {"modhash": "", "dist": 26, "children": [{"kind": "t3", "data": { "approved\_at\_utc": null, "subreddit": "aww", "selftext": "", "author\_fullname": "t2\_asw2a", "saved": false, "mod\_reason\_title": null, "gilded": 0, "clicked": false, "title": "/r/samoyeds: /r/Aww Subreddit of the Week!", "link\_flair\_richtext": [], "subreddit\_name\_prefixed": "r/aww", "hidden": false, "pwls": 6, "link\_flair\_css\_class": null, "downs": 0, "thumbnail\_height": 140, "hide\_score": false, "name": "t3\_cc1syd", "quarantine": false, "link\_flair\_text\_color": "dark", "author\_flair\_background\_color": null, "subreddit\_type": "public", "ups": 95, "total\_awards\_received": 0, "media\_embed": {}, "thumbnail\_width": 140, "author\_flair\_template\_id": null, "is\_original\_content": false, ... }}]}}

This is just a wall of characters, though we can see both the { braces and [ brackets we associate with dictionaries and lists, respectively. We can see strings are in quotes, there are also numbers and booleans. You can also note that here we can see some of the subtle ways in which JSON is not quite Python. For example, the booleans are written false in lower-case, whereas Python uses False. Similarly, empty strings are written as null, where Python uses None. Despite these subtle differences it should look pretty familiar.

With all of these braces and brackets, it would appear that this structure is nested, but it is hard to see how when it is a wall of text. For JSON we have the notion of ‘pretty-printing’. This is where the text is formatted in a more readable way. The method json.dumps has an argument index = X where X refers to the number of spaces to indent for each level in the hierarchy. Conventionally we use 4 like so:

print(json.dumps(THE\_DATA, indent=4))  
  
> {"kind": "Listing", "data":   
 {"modhash": "", "dist": 26, "children":   
 [{"kind": "t3", "data":   
 {"approved\_at\_utc": null,   
 "subreddit": "aww",   
 "selftext": "",...

Pretty printing can help us navigate the JSON. Recall that the data that we receive is in a structure that is not oriented to data science at the outset, it’s organised towards the system that creates and manages the data. For Reddit, this means that it sends down a huge amount of extra text that would be useful if you wanted to display Reddit yourself. This is what one might do with a third-party Reddit client for iOS or Android. They take this data and use it to format the Reddit page for their app users. We, on the other hand, want to repurpose this data to ask questions *about* Reddit. This means we have to learn a little about how the data is formatted, ask for the correct data, transform it into a DataFrame and then ask questions of the DataFrame.

Since we are just learning to get data into Python at this point, our questions should not be too complicated, but they can still be useful. We will focus on some of the skills learned in the last chapter, like slicing data, counting elements, and getting an average. But first we have to get the data in.

To practice, I have prepared a json file of Muppet episodes from the first four seasons of The Muppet show. This data came from TheTVdb.com. That site is a third party database set up to describe the episodes, characters, summaries, and details of television shows. In a later chapter, we will show how to access this data directly using authentication. For now, however, we will simply use the data provided with this book. If you need to find this data, simple download it from xx.

First we will want to import JSON and then load up the file. When we do that, we can start to explore its structure. Since JSON converts to dictionaries and lists let’s find out which one is the root? If it is a list, we will have to iterate through the elements. If it is a dictionary, we will have to navigate the keys.

In the example below I show first how to read in json and then navigate some of the keys. From this code we discover that the JSON has a dictionary with two keys, links and data. We are interested in data. Under there are 100 elements in a list, which turns out to be the maximum number of entries theTVdb returns in one query. Each element is a dictionary corresponding to that episode with keys airedSeason,writers, overview,seriesId, etc.

import json   
import os   
  
filein = json.loads(open("..{}Data{}muppetEpisodes.json".format(os.sep,os.sep)).read())  
  
print(type(filein)) # This shows it is a dictionary, so let's ask for keys.   
   
print(filein.keys()) # Perhaps we want to explore the 'data' key.   
  
print(type(filein['data'])) # It would appear 'data' is a list.   
  
print(len(filein['data'])) # This list has 100 entries.   
  
# print(filein['data'][0]) # Let's view the first entry. It's very long with a summary and other details.  
  
print(filein['data'][0].keys()) # Inspect the keys - these will go in our table.

<class 'dict'>  
dict\_keys(['links', 'data'])  
<class 'list'>  
100  
dict\_keys(['id', 'airedSeason', 'airedSeasonID', 'airedEpisodeNumber', 'episodeName', 'firstAired', 'guestStars', 'director', 'directors', 'writers', 'overview', 'language', 'productionCode', 'showUrl', 'lastUpdated', 'dvdDiscid', 'dvdSeason', 'dvdEpisodeNumber', 'dvdChapter', 'absoluteNumber', 'filename', 'seriesId', 'lastUpdatedBy', 'airsAfterSeason', 'airsBeforeSeason', 'airsBeforeEpisode', 'thumbAuthor', 'thumbAdded', 'thumbWidth', 'thumbHeight', 'imdbId', 'siteRating', 'siteRatingCount'])

With these 100 entries it seems like we should be able to make a table with 100 rows corresponding to each entry. In theory we could create a DataFrame with the first dictionary as a single row. Then add each new entry just like how we demonstrated adding a row in the previous chapter. However, that would be pretty tedious. Luckily, pandas provides a command for importing json directly json\_normalize (with a ‘z’ not an ‘s’). This function has some quirks and has to be imported separately from Pandas, but is is a really handy command. Notice that it takes in the json once it has already been imported with json.loads. Observe how it then takes the list and reshapes it as a table.

from pandas.io.json import json\_normalize  
  
muppetjson = json.loads(open("..{0}Data{0}muppetEpisodes.json".format(os.sep)).read())  
muppetdf = json\_normalize(muppetjson["data"])  
display(muppetdf.head())

absoluteNumber

airedEpisodeNumber

airedSeason

airedSeasonID

airsAfterSeason

airsBeforeEpisode

airsBeforeSeason

director

directors

dvdChapter

…

productionCode

seriesId

showUrl

siteRating

siteRatingCount

thumbAdded

thumbAuthor

thumbHeight

thumbWidth

writers

0

1.0

24

1

4221

None

None

None

[]

None

…

72476

8.0

1

3549

300

400

[]

1

2.0

22

1

4221

None

None

None

[]

None

…

72476

7.0

3

3549

300

400

[]

2

3.0

5

1

4221

None

None

None

[]

None

…

72476

6.8

4

3549

300

400

[]

3

4.0

4

1

4221

None

None

None

[]

None

…

72476

7.7

3

3549

300

400

[]

4

5.0

3

1

4221

None

None

None

[]

None

…

72476

6.6

5

3549

300

400

[]

5 rows × 34 columns

In the last chapter we would simply display the entire table. This was not an issue since we had upwards of three rows and four columns. But now we have 100 rows and 34 columns. That is too much to print. Nevertheless, it is important to practice *data skepticism*. That is, did it work? So, we should print a little bit of the data using the commands df.head() and df.tail() to see if things worked as expected. These commands print the first 5 and last 5 rows respectively. If you want to print more rows you can use that as an argument as in muppetdf.tail(10) to print the last 10 entries in the DataFrame.

You might also notice that we did not do json\_normalize(muppetjson). Instead, we did json\_normalize(muppetjson["data"]). Try removing ["data"] and see for yourself what happens - it will be one row where all of the data is in a single, very long, cell. *This* is the reason that we wanted to explore the data a little bit first. It seems that json\_normalize will transform JSON into a table, but it will only use the top level keys as columns (or second-level keys, if the value of the key is itself another dictionary, in the muppetjson case these would be links.first and links.last).

TheTVdb is certainly not the only place for data about The Muppet Show. Later, in the chapter on merging, we will look at the data dumps from the much larger Internet Movie Database (http://imdb.com/). This data set is much larger as it contains information about every show and movie on imdb. Thus, it will involve considerable care to properly slice the data down to the cases that we want.

# Markup languages: HTML and XML

A markup language is a language that appends characters to either side of values in order to tell what that value means. This book was actually written in a simple markup language called ‘markdown’ that is used in lots of places including Jupyter notebooks. For markdown, doing something like **make this bold** means that the characters were encased in two asterisks like so: \*\*make this bold\*\*. For markup languages that end in ML, like XML and HTML, typically the content is encased in tags like “<this>” and “</this>”. The one with the / is the ending tag.

HTML is the markup language used all over the web. Sometimes the content is stored on a server in HTML but often times it is stored in a database and rendered as HTML for the user. For example, it should be obvious that Google does not store a unique HTML page for each person for each query. Instead, it has a vast database of links and has algorithms to transform those into a page on demand. The page that is sent to the browser contains primarily HTML but the data came from elsewhere. Such HTML data it might look something like the following:

<html>  
 <head>  
 <title>   
 This is the title!   
 </title>  
 </head>  
 <body>  
 This is a webpage!  
 </body>  
 </html>

If you were to copy that text, paste it into a plain text file with the extension .html and open it in a browser you will see a blank page with the title bar saying “This is the title!” and a single line saying “This is a webpage!” in the plain, default format. Most functional webpages are vastly more complex but many of the ideas are the same.

XML is like HTML but the tags will be different. XML stands for eXtensible Markup Language. This is because XML is just a generic standard but it can be extended in a host of ways depending on the needs of the user. For example, in network analysis, we often use a format called GraphML, which is just XML with a special schema for networks that have special characters for things like a ‘node’ and an ‘edge’. As another example, when you download data from Wikipedia it will be returned in XML format with tags for things like the ID of the revision, the date of the last revision, and the content of the Wikipedia page.

Getting data from a webpage or XML into a DataFrame is considerably more tricky than using JSON. This is because you will have to build the row from scratch. Fortunately, there are a number of ways to ease this process. The one that we will use is with a Python library called beautifulsoup. This library will do a lot of the parsing of a webpage for you. The trick is being able to query beautifulsoup effectively so that you get the data you want. This can then be a powerful way to parse pages and data.

In Chapter XX we will cover some of the complexity of getting data from the web, including ways of getting data from authenticated sources. For now, however, we will gloss over much of the details and work on webpages and XML documents that have been saved for you. Both the webpage and the XML documents are available online free of charge.

## Basics of beautifulsoup

BeautifulSoup takes in a blob of markedup text and parses it for use. When you use the library it is convention to call the parsed text a soup. We use a soup to help us find text that could be anywhere on the page. Since XML and HTML documents are hierarchical, if we did not have this ability we would have to navigate through the hierarchy. In the above example of HTML, getting the text from the title hierarchically would be soup.html.head.title.text however, the soup knows that title is a tag so you can just ask for soup.title.text and it will return "This is the title!". This ability to just look for tags is especially useful for things like looking for links (which all start with the <a> tag, as in <a href="www.duckduckgo.com">Search with DuckDuckGo</a>.

To look at beautiful soup, let’s first have a look at an HTML page and then an XML data file. We are going to use Wikipedia in both cases and use the exact same page so we can see the difference between the data as seen on the web and the data as seen in HTML. Both files refer to the “Canada” page on Wikipedia. Simply download these two files and place them in the Data folder for your code to run properly.

## Wikipedia as a data source

In my courses and my research I lean a lot on data from Wikipedia. It is truly a marvel of the Internet age. The accuracy of pages on Wikipedia is often high caliber and the data that is available from the site is often staggering in its depth. In past work I have made use of Wikipedia pages, pages for authors, statistics for page views and edits, pages in different languages, and more. In research I like to suggest that Wikipedia is a great place to start but a terrible place to end. This means an emphasis on critically engaging the content as well as checking out the sources.

One of the nice things about Wikipedia is that as a freely accessible encyclopedia, there’s always content that can be used in teaching and research. In this chapter we will use a snapshot of a Wikipedia page that has been stored in the Data file. We will compare that snapshot as formatted HTML as well as unformatted XML with wikitext. In later chapters, we will revisit Wikipedia for plotting temporal data as well as practising one of data science’s most slippery techniques: the regular expression. For now, we are going to use beautifulsoup and some pandas methods to simply explore the structure of a Wikipedia page and get some basic statistics about its features. This will simultaneously show how to work with some html data and how to build a DataFrame.

## Wikipedia as HTML

On the web, Wikipedia is formatted as HTML. It has links that go both within Wikipedia as well as links that go to other sites. The page will have a consistent format regardless of the Wikipedia entry. You can see the underlying text that we are working with by opening Canada\_Wiki.html in a text editor, or see it formatted by opening it in a web browser. The page should look similar to https://en.wikipedia.org/wiki/Canada although the live page will undoubtedly have at least a few tweaks to the content between when the book was published and when you look at the page.

import bs4,os   
  
wikiHTML = open("..{0}Data{0}Canada\_Wiki.html".format(os.sep),'r').read()  
print(len(wikiHTML))

940053

At this point wikiHTML is just raw text. Printing the length shows it is a long series of characters, so it is probably the page as expected. We can preview the text by printing a range of characters such as print(wikiHTML[:200]) for the first 200 characters. This gets as far as showing the title of the page is Canada. So far, so good.

print(wikiHTML[:200])

<!DOCTYPE html>

Canada - Wikipedia

document.documentElement.className=document.documentElement.clas

## Using BeautifulSoup

We Beautifulsoup to parse the HTML for us. This means that it will create a ‘soup’ which is an object that allows us to query for things in the HTML hierarchy without necessarily going step by step. Recall above in the simple HTML example that the text for the title was nested <html><head><title>? Well with a soup you can just ask for soup.title.text instead of soup.html.head.title.text. This is pretty handy for sourcing links and sections from an HTML page. See below how we first parse the page, print the title text, and look for links.

# Step 1. Make the soup   
soup = bs4.BeautifulSoup(wikiHTML, 'html.parser')  
  
# Query the soup  
print(soup.title.text)  
links = soup.find\_all("a")  
print(len(links))

Canada - Wikipedia  
3920

This procedure came up with 3920 unique links in the HTML page for Canada. Yikes, even for a single page on Wikipedia, we are already getting into scales that would be hard for people to work with on their own. Imagine trying to count all those links by hand, for even a handful of versions of the page.

Although it says that there are 3920 links on this page, I think we ought to practise some data skepticism here. Are URLs the only links available? One way to check is to parse through them and check that the result of the link contains both href (“hypertext reference”) in the HTML tag and http in the URL string. If they don’t contain href then they are probably either accidents or links to different sections within the page.

urls = []  
internal\_links = []  
  
for souplink in soup.find\_all('a'):  
 link = souplink.get('href')  
 if link: # That means the link is a hypertext reference and not a section heading  
 if 'http' in link:  
 urls.append(link)  
 else:  
 internal\_links.append(link)   
 else:  
 print(souplink)  
  
print(len(urls),len(internal\_links))

<a id="top"></a>  
<a class="mw-selflink selflink">Canada</a>  
<a class="mw-selflink selflink">Canada</a>  
<a class="mw-selflink selflink">Canada</a>  
919 2997

So we have discovered here that the page contains 919 links that have http and are thus likely to be external links elsewhere. There are also a whopping 2997 links to somewhere else either on the page or within the same domain. We can have a preliminary look at the internal links to see what sort of data we have.

for i in internal\_links[:10]: print(i)

/wiki/Wikipedia:Featured\_articles  
/wiki/Wikipedia:Protection\_policy#semi  
/wiki/File:En-Canada.ogg  
#mw-head  
#p-search  
/wiki/Canada\_(disambiguation)  
/wiki/Geographic\_coordinate\_system  
//tools.wmflabs.org/geohack/geohack.php?pagename=Canada&params=60\_N\_95\_W\_  
/wiki/File:Flag\_of\_Canada\_(Pantone).svg  
/wiki/Flag\_of\_Canada

This shows that the links appear to be pointing to a variety of locations and file types, including links on the Canada page itself, audio files, links to other relevant pages, etc. Presently, I do not have a specific research question about these links, but we can come up with one by thinking about comparing these links across pages. For example, some ideas might be to ask: - Between countries, which have more links to external government websites? - For the same country in different languages, are there similar levels of content? In this case, looking at Canada in English and French (the two official languages) would be revealing. - Over time, has the number of links increased? Did it level off after a certain point?

These are left as puzzles for you, and more will be available in the exercises. But to get you started, I will show here a little snippet for how to get these links into a table and then use map/lambda to detect which links are to internal wikis.

wikiLinks = pd.DataFrame(internal\_links,columns=["internal\_links"])  
  
def get\_wiki(text):  
 if text[:5] == "/wiki": return True  
 else: return False  
   
wikiLinks["wiki"] = wikiLinks["internal\_links"].map(lambda x: get\_wiki(x))  
wikiLinks.head(10)  
  
print("There are {} internal links on this page, {} of which are unique, and {} of which are to other wiki pages".format(   
 len(wikiLinks["internal\_links"]),   
 len(wikiLinks["internal\_links"].unique() ),  
 len(wikiLinks[wikiLinks["wiki"]]) #Notice here I sliced to only "wiki" == True.  
 ))

There are 2997 internal links on this page, 2355 of which are unique, and 2035 of which are to other wiki pages

## Using Pandas read\_table method

Before moving on, I wanted to briefly illustrate how parsing HTML can be a joy and a curse. The pandas read\_table method uses beautifulsoup to find tables in an HTML document and then parse them. However, it does not work great. Looking at some of the ways it did not work out-of-the-box can help you appreciate some of the challenges that lie ahead in properly parsing data. Don’t get me wrong, it is pretty nifty and with some tweaking you might get it working for you, but these are the sorts of things that highlight how this work can be messy and requires significant finesse.

Applying read\_table to the Canada page gets us a table that corresponds to the infobox in the upper right corner. Unfortunately, the parser by default seems to struggle with many of the other tables on the page.

soup = bs4.BeautifulSoup(wikiHTML,'lxml') #res.content  
tables = soup.find\_all('table')[0]   
parsed\_tables = pd.read\_html(str(tables)) # This will return a list of DataFrames, one for each table detected.  
print(len(parsed\_tables)) # This will show us there is only one table detected.   
display(parsed\_tables[0].head())

1

Canada

Canada.1

0

Flag Coat of arms

Flag Coat of arms

1

Motto: A Mari Usque Ad Mare (Latin)"From Sea t…

Motto: A Mari Usque Ad Mare (Latin)"From Sea t…

2

Anthem: “O Canada”[a]

Anthem: “O Canada”[a]

3

NaN

NaN

4

Capital

Ottawa45°24′N 75°40′W﻿ / ﻿45.400°N 75.667°W

So showing this we can see that the data is still really messy. The column headers have the names “Canada” and “Canada.1”, there is repetition in the text and some empty text. In the 5th row (index[4]) we see Capital in one column and then “Ottawa45°24′N 75°40′W﻿ / ﻿45.400°N 75.667°W” in the next, meaning that it did clearly did not record a return character between Ottawa and a geocode. This is a great example of why much of data science is actually wrangling. The data you know is there, but it is in a messy format and needs to be cleaned.

In the next chapter we will look at some approaches to cleaning and detecting different features in the data. We will use regular expressions to see if we can detect URLs directly as well as other features like the latitude and longitude coordinates seen above. In the meantime, we will look at the raw XML for this page as well as a host of other data types.

# XML

XML stands for ‘extensible mark-up language’. XML files can be generic or have a document type. For exmaple, GraphML is really just XML with a specific schema that is used for social network graph types.

Like HTML, XML is a markup language that uses less than < and greater than > to encase the element tags. The text inside these tags must have some special characters escaped.

<start>   
 <middle>  
 <end1> Here is an element! </end1>  
 <end2> Here is an element! </end2>  
 </middle>  
</start>

Elements have an “element tree”. Above, start is the root node, middle is a child and end1 is a child of middle. end1 and end2 are siblings.

XML is a self-documenting style, which means that you can insert details about the elements into the document itself. For example, open up the included Canada.xml file in a text editor.

<mediawiki xmlns="http://www.mediawiki.org/xml/export-0.10/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://www.mediawiki.org/xml/export-0.10/ http://www.mediawiki.org/xml/export-0.10.xsd" version="0.10" xml:lang="en">  
 <siteinfo>  
 <sitename>Wikipedia</sitename>  
 <dbname>enwiki</dbname>

Here you can see the link to the schema. This is a file with a whole ton of details I’ve never really needed in research. These details are, however, important for specifying what is standard for that type of XML (in this case the mediaWiki export). That’s good because it means that well formatted XML should be reliable for it’s type and easy to manage.

Most of the time, we will not be so concerned with the top of an XML document. Rather, we will simply want to navigate the element tree to get to the element(s) that are of concern to us. Sometimes, parsers will already be written which takes the XML and loads it into a data structure for us. This is the case with graphML, the common format for social network data. To load graphML files you can use either the networkx or igraph packages. I mention these more in Chapter XX on networks. Newer Excel files (with the extension ending in x, as in .xlsx files) are also XML. Later in this chapter we will use pandas to parse those directly.

In fact, there is a nice Python library aptly called wikipedia that can make navigating the XML structure easy and allow for direct querying of all kinds of elements. We are not using that library here, however, since we are making use of Wikipedia but this is to illustrate navigating XML.

In the script below, we will use load in XML as a string. Then we will use beautiful soup to navigate the document and return aspects of the XML.

# loading some xml  
import bs4, os  
  
infile = open("..{0}Data{0}Canada.xml".format(os.sep),'r')  
  
wikitext = infile.read()  
  
# Note: In some circumstances, the file is saved as encoded data, in which case  
# use the .decode('utf-8') function on the text. As in:  
# soup = bs4.BeautifulSoup(wikitext.decode('utf8'), "lxml")  
soup = bs4.BeautifulSoup(wikitext, "lxml")  
  
print (soup.mediawiki.page.revision.id )

864119742

## Navigating XML

Navigating XML involves moving up and down or sideways along the element tree. In the case above it was clear that I know where to go for the text I wanted (mediawiki.page.revision.id). In general, however, navigating to the right element is a bit tedious. Some people prefer the use of Python’s built-in ElementTree package. In either case, what you will be doing with your code is navigating a tree structure. Trees tend to use the following nomenclature that borrows from both the natural tree but also the notion of a family tree: - **Root**:The base or primary node is called the root node. - **Parent and child**: A parent is a node that has nodes nested within, like ID nested within revision above. In that case, revision is the parent node and ID is the child node. - **Sibling**: Two child nodes with the same parent. Like how sitename and dbname are both children of siteinfo. - **Leaf**: A sometiems used term to indicate a child node with no children of it’s own.

Below I use beautifulsoup to navigate through the tags so that I can get to the data I want. Normally one would do this and then clean it up so that only the proper working code remains. Notice that even though mediawiki is actually at: ~~~ html

… ~~~

We do not need the full path to access it, similar to how it was done with HTML. BeautifulSoup will return mediawiki by going to soup.mediawiki.text. But also note, that this is not the text on the Canada page. Instead it is the text under that leaf node, mediawiki. To get the text of the page from this schema, we would go to soup.mediawiki.page and get the text from there.

# for i in soup.children: print(i.name)  
  
# for i in soup.html.children: print(i.name)  
  
# for i in soup.html.body.children: print(i.name)  
  
# for i in soup.mediawiki.children: print(i.name)   
  
# for i in soup.mediawiki.page.children: print(i.name)  
  
# I discover that we can just say soup.page and it will get the text.   
y = soup.page.text  
  
print (soup.page.text == soup.html.body.mediawiki.page.text)  
print(y[:100])

True

Canada 0 5042916

864119742 864118763 2018-10-15T06:33:55Z

Moxy 8729451

/\* Government and politic

At the moment there is not much to do with this text. We can probably split it up or count the number of characters. Perhaps you could compare the length of text for Canada to other countries. In the exercise I show how to download this data directly from the special export page. But counting characters will only get us so far in answering questions. In the next chapter we will start parsing this text and adding it to DataFrames. Then in later chapters we will look at including even more data in our DataFrames by comparing data from different topics, sources, accounts, or time periods. First, however, we should look at a couple more data structures. The next one, CSV, being one of the most common formats around.

# CSV

Comma-separated values is a common data storage format. Yet, despite it’s name it is actually an amalgam of a variety of possible formats, some of which do not even use commas as separators! Rather, when I say CSV here I’m referring to data that is stored in a rectangular format, with rows, columns, and often headers. Such data is extremely common on the web when looking for data sets. However, lots of tables vary in their specification. Here are some important features to consider:

* **Headers**: Does the file have headers for the columns? In the pandas pd.read\_csv() method (or Python’s built-in csvreader) you can indicate that the file has a header with the headers=True argument. In neither case, however, does it deal nicely with two line headers. IF you have a two line header, you might be better off reading the file in and then writing the file back line by line but excluding the first line. Then using this new file, you would import it as usual.
* **Quote character**: The quote character is used to indicate that all of the text inside quotes refers to one string. This is important in the cases where you are parsing a comma-separated file, but the file has commas in it. For example, imagine a file of addresses with headers: Name, Location, User. Now imagine one line of data is “Drake, Toronto, Canada, False”. It might parse that as Name: Drake, Location: Toronto, User: Canada, followed by a parse error. If the data, on the other hand was:

Drake, "Toronto, Canada", False

Then it is clear that the comma in between Toronto and Canada belongs in the string. Now you might be wondering what if you want to use the quote character in the string? This is where we would use ‘escape codes’. In this case using " to include " inside a string. See this entry for an example:

Name, Location, User  
"Sean \"P. Diddy\" Combs", "New York City, NY", False

* **Delimiter**: Even though it is a comma separated file, sometimes people will use a different way of making the separation. A common option is the tab character (and sometimes it is even referred to as a TSV or Tab-separated values".
* **New Lines**: There are two issues with new lines that sometimes trip people up. The first is that, particularly on some old files, the end of a file has a \r\n rather than just an \n. This is because the \r represents a ‘return carriage’, that is , the cursor should go down one line and return to the left of the screen. This is very similar to a typewriter. Thankfully it is now really rare and almost all CSV files use \n. The second issue is how many \n characters are at the bottom. Sometimes if there is more than one the computer gets confused because in between them it would expect a full row.

It is not difficult to build your own CSV parser. In fact, that is one of the exercises in this section. However, it is clear that there are enough little details to attend to that it makes sense to use the build in packages where possible. Python offers two main ways to parse CSV files. First is the csv library. This is a standalone library that can be imported. It has many options for separators and whether there’s a header. It also has some nice ways to index the data. For example, if you want to store your data as a dictionary with the header as the key and the column as the values, you can use csvreader to do that.

## Using the build-in CSVReader

The basic usage, however, is to iterate through a file line by line. Instead of iterating through with ‘readline’ and splitting the text that comes back, you create a “csv reader”, and this iterates line by line returning not a string of text, but a list split at every comma (or user-defined separator).

import csv,os  
  
with open('..{0}Data{0}MuppetsTable.csv'.format(os.sep), newline='') as file\_to\_read:  
 filereader = csv.reader(file\_to\_read, delimiter=',', quotechar='|')  
 for row in filereader:  
 row = ["{:<20}".format(x) for x in row]  
 print("".join(row))

﻿Name Gender Species First Appearance  
Fozzie Male Bear 1976  
Kermit Male Frog 1955  
Piggy Female Pig 1974  
Gonzo Male Unknown 1970  
Rowlf Male Dog 1962  
Beaker Male Muppet 1977  
Janice Female Muppet 1975  
Hilda Female Muppet 1976

The nice thing about csv, particularly when not using pandas, is the use of the DictReader. This returns a dictionary with the header as the key and the values in that row as the value. If there’s no header line, you can specify a list to be the keys using the fieldnames argument, such as fieldnames = ["Name","Location","User"].

## Using the Pandas CSV reader

To import into a DataFrame directly using pandas, observe:

import pandas as pd  
  
df = pd.read\_csv('..{0}Data{0}MuppetsTable.csv'.format(os.sep))

Just like the CSVReader, the pandas pd.read\_csv method has many arguments for things like headers and delimiters.

TIP: **Using help()**

You can use help in jupyter or in a python console by encasing any method or function in help(). So to learn about all the arguments for read\_csv, you would run help(pd.read\_csv). A word of caution: Note that this is without the () after read\_csv. If you put those parentheses inside the help method, then it will first *evaluate* read\_csv() which means you will be asking for help on whatever read\_csv() returns, not on read\_csv the method itself.

Since we will almost always be moving data to a DataFrame this is often a very handy thing to get working. However, as data gets larger, reading straight into a DataFrame gets increasingly slow. For very massive files you might want to read them piecemeal. How large are we talking? On modern computers we might be talking CSVs in the hundreds of megabytes or more. Below that, Python should be very speedy importing and parsing CSVs. Above that and you will want to consider whether to just read in parts of the file at a time or another strategy, typically to divide and conqueror the data. For really big data (on the order of gigabytes or more, where you will have more data than RAM) you will want to turn to server-based solutions outside of scope in this book, such as Google’s BigQuery.

# Excel

Excel is the popular spreadsheet program from Microsoft. Files can be stored as either .xls or .xlsx. The original .xls is a bytestream proprietary file format, but the details are handled by PANDAS. The second one was published as an open standard and is in fact a wrapper over a specific format of XML.

In general, we simply want to import a sheet with:

<sheet> = pd.read\_excel(<file\_path>)

However, I strongly encourage you to check with the documentation. See: https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read\_excel.html You can also see in this document the trials with trying to remember function names. They’ve deprecated the ‘sheetname’ argument, to use ‘sheet\_name’ whereas ‘skip\_footer’ has been deprecated to use ‘skipfooter’. So there are many useful arguments there, but they will need a little patience for complicated sheets.

Excel documents can be pretty complex. Pandas will read the first sheet, but not the others unless specified. It also is not great with headers as can be seen in the worksheet examples. Your mileage may vary, so remember **data skepticism**. Review your data and check it thoroughly before working with it.

import pandas as pd, os   
  
mt = pd.read\_excel("..{0}Data{0}MuppetsTable.xlsx".format(os.sep))  
display(mt)

NameGender

Species

First

Appearance

0

Fozzie

Male

Bear

1976

1

Kermit

Male

Frog

1955

2

Piggy

Female

Pig

1974

3

Gonzo

Male

Unknown

1970

4

Rowlf

Male

Dog

1962

5

Beaker

Male

Muppet

1977

6

Janice

Female

Muppet

1975

7

Hilda

Female

Muppet

1976

# Serialization

Sometimes, you want to close a program and pick up right where you left off. This might mean ensuring that all the objects are in the state that you want them to be with no further processing. This process of creating a file that will represent the state of some values is called **serialization**. We ‘serialize’ variables or data structures. Now, Python being Python, they had to give it a more friendly name - pickling.

One useful approach with pickling is when you are processing text on a server and you are doing something complicated, you can pickle all your current state of things if the program goes sour, then pick up where you left off. The nice thing is that you can put all your variables, objects, etc. in a collection and pickle the collection without worrying about the shape of it. You can only serialize one object at a time, but of course that object can be a collection of numerable other objects.

Since these files are meant for the computer, they might not make complete sense read as text. But we can save them and read them into a new variable later. This is done with the following syntax:

import pickle   
x = <object>   
pickle.dump(x,open(<file>,'wb'))

And to load the object again (with any name): ~~~ python y = pickle.load(open(), ‘rb’) ~~~

import pickle  
  
x = ['1','2']  
pickle.dump(x,open("temp.txt",'wb'))  
y = pickle.load(open("temp.txt",'rb'))  
print(y)

[‘1’, ‘2’]

## Example use case of pickling

A while back I was processing some Twitter data. One thing I wanted to do was to build egonetworks of users. This would be a social network of a user, the user’s followers and crucially the connections between these followers. The algorithm for doing this is a bit tricky. But the important part was that it was slow. You had to more or less look through all the followers of each of an account’s followers. This is ok if you look through my few thousand followers or those like many of my colleauges. But every now and then someone would follow Obama or Katy Perry and suddenly you have to download millions of followers. Times like this a program can go sour. So when it does, I often wanted to pickle the state of the progress so I could start up where I left off, rather than start at zero.

To make a program for collecting data more robust, I did the following: 1. Catch the ConnectionError event 2. call a custom pickleProgress() method that dumped all the progress inside a file. 3. The file would be timestamped for the file name. 4. I would retry the connection and if that didn’t work the program would shut down. 5. When restarting the program, I would check if there was a progressPickle file and I would take the most recent one and pick up where I left off.

## Pickles can expire: Check the version number

Notice that we are using 'rb' and 'wb' with the pickles. This is because Python 3’s default pickling version writes the pickled object as a bytestream rather than as a series of characters like the text you’re reading here. We will discuss bytestreams more in the next chapter when we look at character encodings. In short, bytestreams are strings of characters as readable by a computer, not encoded as text for humans. Between Python 2 and 3, pickles went from being regular characters to bytes, so you cannot read a Python 3 pickle using the previous Python.

Because pickles are so tightly coupled to the specific version of Python (and the libraries installed even), they are really handy for short term storage but too fragile for long term storage. Instead, one should use one of the file formats discussed above, such as CSV / XML / JSON or even Excel which has extensive support and care with backwards compatibility. If you find yourself with really demanding storage needs, you will probably want to seek out extra resources on this. One example would be to look into the feather package. It was co-written by the creator of Pandas, Wes McKinney, is very fast, compact, and scalable. But it also suffers from the same versioning issues as pickles. Regardless of file type you choose, remember to check both writing the file to disk and re-reading it again before you put it away for a while.

# Conclusion

File formats might not be the most exciting topic and certainly one that is often considered far away from traditional social science, or so one might think. In practice, I certainly remember in graduate school the trials of getting data for SPSS and having to convert it to Stata or SAS. Prior to the massive rise in the use of Python and R, quantitative data in social science was almost exclusively done using programs that were 1. for pay, 2. unorthodox syntactically, 3. incompatible. What can be seen from this chapter is actually that Python is for free, that the file formats are not software specific, and that it is assumed that Python should be able to open the data. This is great news not just for data science, but for science. In general, we want science to be as open as possible. Obviously, some data must be restricted for reasons of privacy, but the norm now is towards being less locked into a product. Using Python (or R, for that matter) involves working on a product that is community-driven, open source, freely accessible, and meant to abide by many established conventions in computer science/programming. As we saw at the very end when looking at pickles, when we have to return to a language-specific file format we are right back where we started with version incompatibilities. The pickle thus is not really meant to be a long term storage format, whereas XML, CSV, or JSON can get that job done.

The goal of this chapter was to get you acquainted with the logic of many of these tools. The exercises for this chapter will go into a little more effort as we ask just a little more of you than was on offer in the main text. But beyond these exercises, when you start to explore data yourself, you will definitely encounter all manner of contingencies outside of what was mentioned in here. Corrupted files, strange delimiters, and files that are too large to load are all a part of dealing with data science. In a perfect world, all data would be cleaned up, well formatted, and pre-processed.

Social science often dreams of that perfect world in a potentially dangerous way. An emphasis on survey research and qualitative coding of transcripts really suggest that claims are made with data of a specific shape and size, rather than data more broadly. We might be inclined to call this the ‘independent case model’. It has each row as a case and columns to represent variables. It looks a lot like a DataFrame, and for good reason. The tabulation of cases allows for all kinds of statistical routines that otherwise are not as accessible or tractable. What I am suggesting here, however, is that the process of transforming social life into this table does not have to happen as a part of data collection. The data can be collected from a variety of sources, in a variery of ways. Granted, this can affect generalisability. Independent random sample data collection is still an excellent way to make a generalisable claim. It is instead, to suggest that the drive to make claims generalisable in the mindset of random samples from a large population can sometimes unduly restrict our ability to make claims about a specific population or group. Having all of the comments from a message board or all of the pictures posted in a forum means we can make very extensive claims about that board. Stated differently, we start with the data in the shape it comes in and then work on transforming it into a DataFrame. We do not start by looking for places where life imitates the DataFrame and try to come up with questions to ask.

As we step outside of what we can do with survey research we will find that there’s all kinds of ways of creating and managing data for the purposes of making claims. To reiterate some of the things said in the introduction, we are learning to build socialscopes here, not just take whatever scope was presented at the outset. In the next section we begin to embark on this process of reshaping data so that it can meet our needs. We will have to extract features from text, merge different data sets together, come up with ways to aggregate data (for example, to get averages per day, or within state). This exploratory process will help us learn a little about the shape of the data, what we might be able to expect from it and what sorts of claims we want to make. Data exploration is a journey, and with these tools for getting data into DataFrames, we just got our first vehicle. Let’s see where we can go.