Created by Elizabeth Wickes for the 2023 Text Analysis Pedagogy Institute, with support from Constellate.

For questions/comments/improvements, email wickes1@illinois.edu.

Workshop context

- purpose of this of our shop is to sort of lay a foundation of what web scraping is
- understand the kinds of situations you may find
- review the problem solving approaches
- understand the commonly used technologies and when to use them

Leading into the next workshop, where we focus on exploring xpath and regular expressions. Why are these separate workshops? Those can be pretty heavy to learn and sometimes you really don't need them. Sometimes things like Google sheets can take care of your needs. The whole purpose of this is to explore the breadth of problem solving options available for web scraping so you can pick the one right for what you are working with.

What this is not going to be

To make it quick, the text you want needs to be actual text somewhere on the website.

This means we aren't going to be talking about:

- extracting text or other content from PDFs
- OCRing data/etc from images
- deep dive into working with APIs
    - although we will talk a bit about APIs

What I love about web scraping

Aside from getting data you may not be able to otherwise...

There are so many interesting situations that involve some deeply creative problem solving strategies. You have to be a little bit like a private investigator. Sleuthing through the website trying to figure out if there are structures in the pages that we can use to our advantage.

The hierarchy of your time

Your time and research time is important. Same for the people you might be teaching these things to. When you're first learning about a lot of programming and other tools sometimes your first thoughts are, hey let me write a script for this! Sure, getting more practice can be good, but you've got to respect your own time and needs.

Where does this come into a project?

Generally you should already have some sort of question or ask your learners if they have their research question or area. Web scraping comes in during the phase of data gathering or data discovery.

And this workshop sort of presumes that you have data somewhere that you can put your eyeballs on and say, I need that data. And once you got to that point of like, okay, the data exists. How do I get it?

The first question about web scraping is not about web scraping

Always check first, does this data exist somewhere in a more accessible format? I've done a lot of scraping for things in the past but now other people have put datasets from those things up.

Pro: you can download a thing and it's already data!
Con: it may be an older snapshot, may not have all the things you want, etc.

However, this may be enough for a small proof of concept. Don't discount something quick and easy just to explore the vibes!

So where can this data be found?

Now, this isn't a workshop about how to find data. However, when you are teaching these workshops, this might be a good place to advertise your services etc. Here are some of my recommendations (this is going to be pretty US specific):

1. https://www.re3data.org/
    1. Repository of repositories, you may be surprised by what's already out there!
2. https://commons.datacite.org/
    1. Search metadata for all DOIs registered under datacite, which does contain a good amount of data! Has cool stuff, but the results can be a bit noisy
3. https://www.icpsr.umich.edu/web/pages/ICPSR/index.html
    1. Lots of social science data, amazing search engine
4. https://data.gov/ or local city/metro area data repository
    1. Government data etc
5. Others that are....less of my favorite
    1. https://www.google.com/publicdata/directory
    2. https://www.kaggle.com/datasets
6. Also search if there's an API, and sometimes you need to search for that specifically

Other important factors to stress:

1. Check your local resources!
    1. Subject librarian or other library resources. Be sure to ask directly if there might be something available. Not everything makes it to the website perfectly.
2. Lots and lots of googling
    1. Sometimes you can find stuff in odd places or there are little personal data collections that people may have online.
    2. I like to give myself a good hour or so to dive deep to try and find something, but I usually have to set a timer so I eventually actually stop...

So eventually you're to a point where you can put eyeballs on the data in front of you, it's on a website, you can copy/paste it, etc. However, you can't find it as a nice pretty dataset to download. This is where web scraping fits in.

Briefly, how web pages do the thing

(Very briefly, very high level)

Web pages use HTML as a markup language to dictate how the content should be displayed. Headers, bold, etc. It also allows for things like hyperlinks, pictures, and other content to be displayed. This markup is also just text in specific formats.

Web browsers "parse" or read this text and attempt to understand the content versus the structure of the HTML given to it. The web browser then uses it's tools to display it for you.

For example, `<b>This will be bold text.</b>` will display that text as **bold**. You won't (or shouldn't) see the `<b>` tags displayed, just the text rendered as bold.

Many websites have tons of pages and content. These are likely not saved as separate individual files. The HTML to display those many pages is usually coming from some form of a template. The tool reads the data from the database, and sends it to the template, the template, then generates the HTML with the contents to be displayed. This is a really simplified way explaining it, and there are many many tools that do these sorts of things.

The important thing to remember is that when you're dealing with , is that the data is being stored somewhere with some structure. You won't always know how or with what, but you can usually get a good idea about it by looking at the HTML being generated to display it. The HTML formatting elements will often have content specific tags about what it is displaying. For example if it's about a person, they may want the name displayed in a certain way. There are special ways to define HTML formatting where you can label specific groups of content, like a name. So when you look inside the HTML, you may see the name surrounded by some formatting tag that literally says "name". Not always, but generally, you can use these formatting labels to quite clearly extract the data that you want.

The core tools

We'll be talking first about simpler techniques, as they often will enough to get you going. Then move into the larger tools, like using requests to download things,

pathlib to manage the files, and then using regex and xpath to extract information.

- `requests` a very popular and well supported library for handling http calls. Commonly used to read and download website content or files.
- `pathlib` a modern object oriented way of handling files in python, managing folders, etc. Very much an essential tool even if it lacks some of the big buzz words.
- `re` this is the Python regular expressions module. It is used to match text patterns within free text
- `lxml` this package is used to help parse xml and html files and what we will use to execute some xpath queries
- `bs4` this is the beautiful soup package, and it used for cleaning up messy html. It can be used to extract content if you want, but xpath queries are more powerful
- `csv` and `json` these are two python packages we will use to export our data out

Simpler forms of web scraping

In many cases, you may just want to copy an HTML table into something that is actionable data. This may be a CSV file, Excel file, or maybe some thing that you read directly into Python. There are a variety of tools to help you take a single HTML table and get it into one of those things.

Copy/paste it into a spreadsheet

A simple HTML table

- http://www.neoperceptions.com/snakesandfrogs.com/scra/ident/names.htm
  - Open the page and look in the view source.
  - Looking at the table we can see this HTML comment

```
<!-- The following table was generated by the Internet Assistant
Wizard for Microsoft Excel. -->
<!-- ---------------------- -->
```

```
<!-- START OF CONVERTED OUTPUT -->
<!-- ------------------------ -->0
```

So this content is likely not being served up to us by another tool, but this is just plain HTML. Say that we want this back into a data file.

We can select the text within the table, copy, and paste it into Excel or Sheets. However, note that the styling will also get pasted in.

More complex table

Let's look at another one. https://threadcolors.com Looking at the scale of this and the HTML all being horizontal, we can safely say that something is generating this html. You can also see the javascript stuff in there as well.

Let's copy/paste the first part of this table in and see what happens. On my computer, Excel doesn't paste the colors, but google sheets does. I've also seen pictures and other things get pasted in, along with font styling and other formatting.

Suppress styling with "Paste special"

Excel and Sheets have versions of "paste special". There are some really nice extra tools in here if you've never explored. I'll briefly explain where to find these things, but interfaces and versions always change.

- Excel has a few places for finding this. I like to right click on the top left cell where it should go, then select "paste special".
  - You'll see some options there, including another Paste Special. Choosing that opens up a window where you can choose HTML, Unicode text, or text.
  - Generally I choose "text" to get just the plain text.

Microsoft and Google each have data import tools etc. you can also play with.

Google sheets importing tools

Type `=IMPORT` into a google sheets cell and you'll see a bunch of options. These include tools for reading in data files online etc. Let's look at `importhtml`.

https://support.google.com/docs/answer/3093339?hl=en

Functions like these can be really handy, but you need to work with them really closely. These functions can work really nicely, but expect to spend some time playing with the arguments to ensure it's working correctly.

This will take a URL along with other arguments and import the specified table of data into your sheet.

The second argument is labeled query, but is asking you to specify if they should be importing a list of data or a table of data. We want to specify table.

The index argument is asking you to specify which table on the website, it should import. Some pages may have dozens of tables, so youcount from the top down and provided the number (starting at 1) for which table it should be. Always check to ensure the right one has come in, because the way the tables appear on the website may not exactly match other specified in the HTML if there is this that many tables on the page that you may accidentally hit the wrong one.

Here is our cell argument:
`=IMPORTHTML("http://www.neoperceptions.com/snakesandfrogs.com/scra/ident/names.htm","table",1)`

Take a moment to look at how the data has been read into your sheet. The upper left cell, where you originally put the function information, will still contain the function content, but the other cells will only have the actual text contact. You may also noticed that some of the formatting, like bold and italics did not come through correctly.

Tools like this can extend how powerful this idea copy and pasting into a spreadsheet can be. Say you had a single table with basic formatting, and wanted to import it into a spreadsheet regularly. Using a function like this would allow you to incorporate some amount of automation, where this can be repeated without having to copy paste the table each time. However, there are several limitations. You shouldn't use this without observing the results in case the original website structure has changed. Timestamp access information is also not retained.

The list option is also quite interesting. However, it is only importing the top most list that it sees. Meaning that navigating a deeper structure can be harder.

```
=IMPORTHTML("https://en.wikipedia.org/wiki/Lists_of_American_universities_and_colleges", "list", 6)
```

TIPS: you'll also see `importxml` in this list, allowing you to do easy xpath queries. This won't give you all the power of using xpath with python, but is still extremely useful. Xpath will be covered more later on.

```
=IMPORTXML("https://en.wikipedia.org/wiki/Lists_of_American_universities_and_colleges", "//div[@class = 'mw-parser-output']/ul/li")
```

Looking at the results, we can still see some limitations.

```
=IMPORTXML("https://en.wikipedia.org/wiki/List_of_colleges_and_universities_in_Illinois", "//table[contains(@class, 'wikitable')]/tbody/tr/th/a")
```

If you play around more you can get more, but generally this won't be quite enough.

Summary

In the section, we saw a few tools to copy and paste or import contacts in from webpages or tables. These tools tend to be really useful for smaller tasks or exploration, but can't (okay shouldn't) be taken much further than that.

Downloading things

Now we're getting into a situation where we have a list of things we need to download.

This exact situation will depend on what you're doing. I've done this where each page had 50 PDF links that I needed to download and there were 20 total pages. You may also have a page with 100 images and you want to programmatically download all of the images on that page. There's so many reasons that you might want to do this, but the nice thing is this sort of task is a great initial web scraping task.

This sort of task also lens itself really nicely to combining both manual work and programmatic work. This situation may dictate which one gets which, but is there a lot of flexibility.

Your first step here is going to identify where is the page that has all of the links on it and then you need to get access to all of them. My general preference is to download and save this page to my computer, because that allows me to experiment with par seeing and figure things out on my own time without having to reload the page or hit their server for every time that I run my script.

Second, you need a proof of concept to check that you can actually get the content out from those pages. Use just one page to experiment.

Tip: you can often be working on these experiments with getting content out while the other pages you need are downloading.

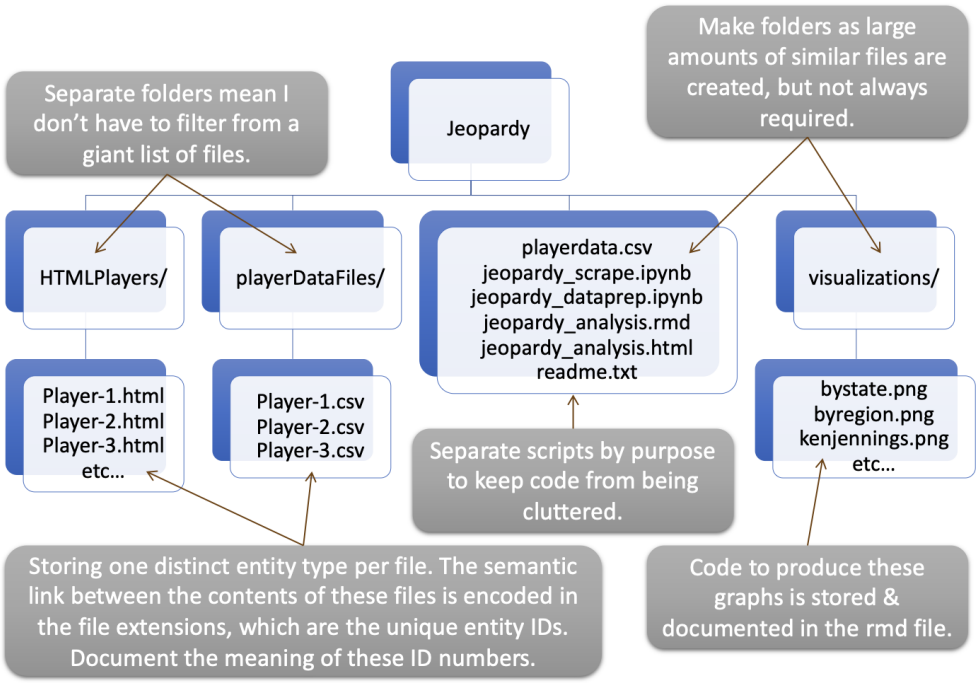
Day 2

Data management and file organization

Some diagrams pulled from previous work:
https://www.ideals.illinois.edu/items/93987

Some themes for organizing your data with web scraping:

- keep the project in a folder, ideally under version control
  - yes things get weird when you start having thousands of files, but that shouldn't stop you
- use folders within that folder to contain things
- be consistent with names
- put meaningful information in the filenames
  - an ID or other content that will easily connect things back to the original source
- retain that meaning between files as you convert them
  - eg use the same identifier information between different versions of the entity



Before we move on, some considerations

There are a few things to consider before we move on where we are programmatically downloading things from someone else's server. Not every website wants to be scraped. Some have restrictions some have blocks, and there's a certain kind of etiquette that we want to follow.

First, we want to keep the speed we are hitting their server to something reasonable. This is usually a minimum of 4 seconds, but I've worked with pages that asked for 30 seconds delays.

Second, some ask that you only do large scale harvesting or scraping during "off peak" times. This often means overnight.

Third, some pages may just completely ban scraping tools from being used. Usually this is because they have an API they'd prefer you to use (and usually pay for) or because the data is sensitive in some way. Let's look a few examples.

- Linkedin has a hard block on programmatic web scraping because their data is really valuable and they want to sell it to you.
- Google will quickly block you from scraping their results because they want to you to use an API. Many of theirs are open and reasonable to use, but they don't

want HTML scraping.
- Archive of Our Own (AO3) has a block against it because they don't want search engines to index the results. This gives them control over story and author information and the ability to fully take things down as needed.

But how can you know for sure? This can be hard and there's no single answer. You can often check the `robots.txt` file for the website. You can read about this file here: https://en.wikipedia.org/wiki/Robots.txt Very generally, it will contain information for humans and for bots, and give you an idea about limitations, etc. Not every site will have it, but most with data will.

- https://en.wikipedia.org/robots.txt
- https://archiveofourown.org/robots.txt
  - my favorite "cruel but efficient"
  - note the crawl delay
- https://www.fanfiction.net/robots.txt

You can ask for permission to go out of bounds for this, especially for research. Just be respectful.

Handling delays

Most programming languages will have some ability to "delay"actions. We will use the `time` module in Python to delay our execution.

`time.sleep(seconds)` takes a number of seconds and pauses script execution for that long. Other languages use `ms` instead, so be mindful if switching!

```
import time

for _ in range(3):
        print("hello!")
        time.sleep(5)
```

Downloading things off one page

Starting with the simplest version for sure, we have one page with a side of links, and we want to download the results of those links. What those files are, doesn't really

matter because you're downloading them to disk.

So what I love about this page is that they just have the sql statement right at the top of the page.

https://calphotos.berkeley.edu/cgi/img_query?where-taxon=Allium+anceps

Let's take a look at the structure here:

- clearly these are coming from a database
- there are multiple pages
- the images are displayed on the page
- there are detail links by each image
- being displayed in a table

Tip: Chrome XPath Helper tool

I like to use this to preview the structure of the elements.

There are a variety of tools you can use for this part! Our basic goal for this is to get URL for each of the pictures. Once we have those collected, we can run through them to download each. I'm going to provide these URLs for now so we can focus on the downloading.

Just a small preview of this xpath we'll be using:

```
//td//img/@src
```

- we can use `//img` to get all the images on the page, but most pages will have other images. Best practice is to include something more specific to disambiguate. This is why I have `td` in here.
- Using `@src` allows me to request that it return the value for the source property
- the URLs for the images appear to have a specific folder structure, which I could have also used to gather them
- the URLs gathered are relative links, meaning that I'll need to build the full URL when I'm doing my pass over them.

Let's open the text file with the URLs and start building those up. As mentioned, these are relative links so we will need to do a bit of editing to get them into the full

pattern. You can check out a link on the main page to inspect what the full URL should be and what the relative links are. Looking a that we can discover that the "base" url should be.

Here's a full link:
`https://calphotos.berkeley.edu/imgs/128x192/0000_0000/1209/2448.jpeg`

And here's the corresponding relative link:

`imgs/128x192/0000_0000/1209/2448.jpeg`

This means we'll need to prepend `https://calphotos.berkeley.edu` before each URL to have the full one. There are several ways you can do this and this is a great time to practice your core Python skills.

Some notes:

- using list comprehension syntax here
- using `readlines` to read it in, which returns a list of strings, each string is a line from the file plus a newline character
- `strip` is needed to take the ending newline character off
- I'm concatenating the base before the url from the line, but note that I didn't include the final / because there's already an opening one from the url.
- This will result in a list of all the urls.

```
with open('source_data/alliumpictures.txt', 'r', encoding = 'utf-8')
as infile:
    # urls = infile.readlines()
    urls = ['https://calphotos.berkeley.edu' + u.strip() for u in
infile.readlines()]
```

Working with `requests`

Let's try something basic!

```
import requests

url = "https://loripsum.net/api/1/plaintext/short"
result = requests.get(url)
```

```
print(result)
```

So what we're seeing here is a sucessfull connection, but not the text. We have to ask about that explicitly from out result object.

We do this with `.text` (no parens!) this will allow us to ask for a variable value within out object (versus calling a function). Some objects just work this way, and we know how to do this by looking at the documentation or a tutorial.

```
print(result.content)
```

But our items are images? What can we do. Just a few tweaks. From python's perspective, we are moving from data that's text to data that's bytes.

`requests` actually has a bunch of ways to handle this, and those methods may be better for larger files, etc. However, for smaller files like ours and the fact that we are using pathlib... we can pretty easily handle this.

Working with `pathlib`

You'll note that I didn't use pathlib for reading in that file. That's okay! Sometimes you don't need to.

`pathlib` is a great module for working with files/folders/etc. For webscraping it is ideal because you can very cleanly handle making folders, checking if things exist, making longer file paths, etc. Honestly, when I started using it vs other tools it was game changing.

We'll be exploring things with `pathlib` as we go, but we do need to cover a few basics.

You create `Path` objects to represent files and directories. Once these are made you get access to special methods for taking action on them or getting information back. You create a file and a folder object the same way.

`p = pathlib.Path(string of path info etc.)`

This returns a `Path` object you'll want to save as an object.

We can use `pathlib.Path('pictures')` to work with our directory and then make the file path objects like `pathlib.Path('filename.jpeg')`. Neither of these things need to actually exist for us to make these objects.

We can use the `mkdir()` method to create a folder, and then use the `/` concatenation operator to combine them.

`pathlib` has two awesome path object methods to write out content:

- `write_text(text stuff)`
  - for text!
- `write_bytes(a bytes or non-text doodad)`
  - briefly, for stuff that isn't text

https://calphotos.berkeley.edu/robots.txt

We have a list of URLs now, so we can loop through those and begin downloading them. There are a few tasks we'll need to accomplish.

- create the file name (from the file name)
- create a directory for the new files to go into
- create the full destination path (target folder plus file name)
- open up the requests connection
- access and write the content
- close the connection
- wait for 5 seconds

This is a lot and we build it up bit by bit.

```
# create the target folder object
target = pathlib.Path('pictures')
# make the directory if needed
# does nothing if already exists
target.mkdir(exist_ok=True)

for u in urls:
    parts = u.split('/')
    last_two = parts[-2:] # grab the last two parts
```

```
    fname = "_".join(last_two)
    # print(fname)
    p = target / pathlib.Path(fname)
    print(p) # this is the full path
    r = requests.get(u) #open connection
    p.write_bytes(r.content) # get content, write bytes
    r.close() # always close your connection!!!
    time.sleep(5) # pause to not anger the server
```

One thing I always check at this point is the file size for everything that has downloaded. When in jupyter on a cloud service, that can be hard, but `!` to the rescue.

```
!ls -l pictures
```

Now, what if we had many or some messed up? Using pathlib is awesome here. We can utilize the `exists()` method to check if the file we are proposing to make already exists.

```
target = pathlib.Path('pictures')
target.mkdir(exist_ok=True)

for u in urls:
    parts = u.split('/')
    last_two = parts[-2:] # grab the last two parts
    fname = "_".join(last_two)
    p = target / pathlib.Path(fname)
    # use .exists to check
    if p.exists():
        print("already done!")
    else:
        print(p)
        r = requests.get(u)
        p.write_bytes(r.content)
        r.close()
        time.sleep(5)
```

So we've seen how to gather some contents from a web site, download a group of things from a website, pause a scraper, and even check if that file already exists.

This may have been about files, but this could also be about pages.

Generating URLs to download pages from

Let's look here: https://calphotos.berkeley.edu/flora/

https://calphotos.berkeley.edu/flora/sci-A.html

Say that we wanted to automatically grab all the URLS for the scientific names. https://calphotos.berkeley.edu/flora/ looking here we can see that the pages all go from A-Z in the URLs. There's other things we can do, we know that it should contain `flora/sci` within the content. We could get all the URLS, filter, and then use that as our list. But let's try generating the URLs.

Often times you'll need to generate numbers or other things within a url. You can use a for loop with `range(number)` to generate a set of numbers

In this case we have this theme of `base + letter + .html`. No, we don't need to make all of these ourselves. The `string` module actually has some fun stuff to keep in mind!
Note that most of the items in this module are variables that you are importing instead of functions. This just means that there won't be `()` after the names.

```
import string

print(string.ascii_uppercase)
```

We can see that we have the letters, let's put it in action. Our url looks like this: `https://calphotos.berkeley.edu/flora/sci-A.html` So hopefully you can see where we might put the letter.

```
import string


for letter in string.ascii_uppercase:
    url = "https://calphotos.berkeley.edu/flora/sci-" + letter +
```

```
".html"
    # print(url)
```

Let's add some code to download and save these pages to disk and in a folder like we did before.

```
import string
import requests
import pathlib
import time

target = pathlib.Path("sci-pages")
target.mkdir(exist_ok=True)

for letter in string.ascii_uppercase:
    print(url)
    url = "https://calphotos.berkeley.edu/flora/sci-" + letter +
".html"
    p = target / pathlib.Path("sci-" + letter + ".html")
    r = requests.get(url)
    p.write_text(r.text)
    r.close()
    time.sleep(5)
```

https://www.crummy.com/software/BeautifulSoup/bs4/doc/#bs4.Tag

Simple queries are pretty nice here!

Our challenge with this structure is the lack of attribute tags. We just have a bunch of table elements that are nested. Now, this makes some things pretty clean. But it can make it hard to select certain things with precision when there are duplicates.

We need to take a look at the structure of the website. What is the content we are after? What makes that location unique compared to the other data?

There's no one single answer here. You have to look at the contents and it'll be something specific for each project. But here are some areas to consider:

- Is the content only in a specific tag?
    - like an image or h1 text

- Is the content in a unique part of the tree?
  - like an image but only the images within a table
- Does the content have a specific style class label or html tag attribute that you can latch on to?
  - like the p tags that are marked for formatting with `class = "name"`
  - This may me semantic, like a name, or something structural that uniquely identifies what you need
- Does the text content within the tag have something unique you can check for?
  - like you want all the `td` cells inside a `tr` where the first `td` cell starts with "Total:". Effectively, you want the contents of a row where the first bit of text starts with "total"
- Does an attribute have a specific value that you can check for?
  - like the `href` for an `a` tag has something specific, as in, you want to check all the hyperlinks but only want the ones that link to a specific subdomain

Each of these situations can be coded up. You can usually use some combination of the selection/extraction tool itself along with core python. How you divide that up will depend on your skills with the tools and how nicely the content will play with them.

Some caveats about Beautiful Soup

I'll be honest, I'm not the biggest fan of using this for extraction. However, the utility is there and for simpler things it's pretty straight forward. We will be looking at xpath queries later, and for highly structured html or more complex queries, it really is much more straight forward.

Another consideration: the searching/parsing of Beautiful Soup is generally slower than what lxml can do. Now, for a few dozen or a few hundred files this shouldn't impact you very much. Use whichever clicks the first for your needs. However, should the number of queries go up into the thousands or millions, you'll want to switch over. Speed may not end up mattering because you can get in and out quickly, and don't need to rerun the results. So this isn't a hard rule. Just keep it in mind.

Extracting things with BeautifulSoup

You can read the documentation here for bs4:
https://www.crummy.com/software/BeautifulSoup/bs4/doc

Some of the lingo on this page may not make a ton of sense to you if you haven't spent some intense personal time in the land of XML or metadata. However, this is where librarian instructors can really shine! Many of us are very used to these kinds of discussions, and you can leverage that expertise to promote your workshops.

The sum of it is, each tag is like a node in a tree. That node will have some combination of parent, child, sibling, ancestors, and descendants. This is how you navigate a tree. There's lots of examples on their page that you can work through, but the best way to get used to it is just to mess around.

Loading an html file into beautiful soup

Let's read in a single file to explore some of these tools. In this code we just want to grab one file to play with from our target folder. This also lets us practice a bit with our core python!

```
import pathlib
from bs4 import BeautifulSoup

first_file = next(target.glob('*.html')) # just a fancy way to ask
it to iterate once

soup = BeautifulSoup(first_file.read_text(), 'html.parser')
```

They always use `soup` as the variable name for the parsed content, so I'm using that to match. I generally suggest you do the same with your own work so things match up with documentation.

From here we can now operate on `soup` in a variety of ways.

Simplest extraction

The simplest query is just to go after all of a single element. Maybe you want all the images on a page or all the links. You can grab those and run filters etc. with their content in regular python if need be. This can be a nice place to start and allows you to avoid some of the query complexity.

You can use `soup` with dot notation and a single tag to grab the first one that the parser sees. This will return the entire element.

```
print(soup.a)
```

```
<a href="/">CalPhotos</a>
```

Yup, this is the top of the page. First one it sees. Maybe this is the one you want? Maybe not. This can be good if you need to start digging around the tree and the first ones coming up are the ones you want. You can also chain these together to go directly at something, presuming that the first one it sees is the one and only one you want.

```
print(soup.body.table.tr.td)
```

```
<td bgcolor="DFE5FA" width="5%">
<!-- uncomment the following line to add a logo ----->
<!-- img align = left border=0 height=130 src = "/calflora/icon.gif"
alt = "CalFlora"-->
<br/>
</td>
```

You can also ask for the content of the tag with the dot notation.

```
print(soup.a.contents)

['CalPhotos']
```

And ask for an attribute's value using dictionary-like notation.

```
print(soup.a['href'])

'/'
```

(this result does make sense, as it is linking back to the main page but with a relative link)

Generally our queries will be more complex than these simple ones, but this core syntax is good to keep in mind because we will use it in conjunction with more complex queries.

Day 3

At this point we've got a bunch of foundational skills:

- downloading things to our computer
- handling files and folders, checking if they exist, making names, etc.
- putting delays in between downloading things
- parsing html with beautiful soup

Honestly, the first parts here of just handling the files, etc. are some of the hardest parts and not often covered in the documentation for the actual parsing tools. Things like beautiful soup, lxml, etc will all presume that you've got the files under control.

Choosing a tool for extraction isn't about finding the "right" one. True, sometimes you'll need a certain tool or something else unique to one particular package. Otherwise, the rest is personal preference.

I happen to have learned xpath before beautiful soup, so I'm naturally inclined to go with a tool supporting that. I also regularly need to parse xml files, which lxml also does well. That's just my experience, and mainly because I happened to know xpath before any of the other tools.

More realistic extraction from BeautifulSoup

In our case we want to examine all the link tags, not just the first ones. We can use `find_all`. They also have a specific section on this you can read more about: https://www.crummy.com/software/BeautifulSoup/bs4/doc/#searching-the-tree

```
print(soup.find_all('a'))
```

This gives a large list of all the things. As mentioned, given that this is all the links there will be extras we don't want. There are two ways we can improve these results:

1) make our extraction query more specific or 2) attempt to filter out the content we don't want.

Both are good strategies to consider. You may not be able to uniquely pinpoint the ones you want within the structure and thus need to look at the content itself, or maybe the content all looks the same and you depend on the structure to disambiguate. Maybe there's a combination of the two!

Let's print this out in a for loop for better viewing.

```
for a in soup.find_all('a'):
    print(a)
```

```
<a href="/">CalPhotos</a>
<a href="/flora/">Plants</a>
<a href="/browse_imgs/plant.html">Browse Thumbnail Photos of
Plants</a>
<a href="/cgi/img_query?stat=BROWSE&amp;where-genre=Plant&amp;where-
taxon=Wachendorfia+paniculata&amp;title_tag=Wachendorfia+paniculata"
>Wachendorfia paniculata</a>
<a href="/cgi/img_query?stat=BROWSE&amp;where-genre=Plant&amp;where-
taxon=Wachendorfia+thyrsiflora&amp;title_tag=Wachendorfia+thyrsiflor
a">Wachendorfia thyrsiflora</a>
(snip)
```

We can see a few things. The results we want all likely have the hrefs starting with "/cgi". We can do this directly by referencing the attribute name (`href` holds the url) and compiling a regular expression. Yes, you can use regular expression fanciness in here but you can also just put in any string to have it try and match that substring. This is also something you could do in core python with string tools.

In this case I've told regex that: the start (`^`) of the string should have `/cgi`.

```
for a in soup.find_all('a', href=re.compile('^/cgi')):
    print(a)
```

We can add a search into this on the actual contents of the a tag using the `string` argument.

```
for a in soup.find_all('a', href=re.compile('^/cgi'), string =
re.compile('var')):
    print(a)
```

This will retain our previous filter but also search the name of the plant for "var".

This function has a lot of power and the documentation provides a ton of detail: https://www.crummy.com/software/BeautifulSoup/bs4/doc/#find-all Caution, if the core python doesn't make sense then the example likely also won't make sense.

specifying more structure

There's lots of ways to fuss over searching the structure in beautiful soup. The nice thing about being able to provide queries with more content is that you don't have to add loops or other complexities in. You can simply say: I want all the `a` tags that are directly under `p` tags. Or something similar.

We can do this with some nice shorthand with the `select()` soup method.

This method allows you to specific css structure to select things if you want, but you can also specify more tree structure. This is what we'll be using.

You can specify multiple tags here and it will look for those tags within the tree.

We could say:

```
soup.select("p a")
```

This would find all `a` tags that exist anywhere inside a `p` tag. (xpath equiv: `//p//a`)

We can also say:

```
soup.select("p > a")
```

Where the `a` tags must be direct children of `p`. (xpath equiv: `//p/a`)

We can also combine these:

```
soup.select("table p > a")
```

Saying to select all `p` elements anywhere inside of a `table`, and then an `a` tag if directly a child of `p`. (xpath equiv: `//table//p/a`)

Handling the results

These results all give you a list of tag objects you can further mess with.

We can ask for the contents of the tag:

```
for a in soup.select("table p > a"):
    print(a.text)
```

This will give us all the species names. Let's note that the species name is the only thing in the hyperlink. We also want the number next to it. To get this we need all the `p` tag text, but we don't want all the `p` tags. We can accomplish this by navigating the tree more: find all the `a` tags we want and then ask for the parent tag's text. (this is weird but more common than you think). xpath equiv: `//p/a/../text()`

```
for a in soup.select("td > p > a"):
    print(a.parent.text)
```

`a.parent` is a relative lookup and "becomes" the `p` tags we want. Then `text` is applied to that.

We don't want `a.parent.contents` because that will also return the full `a` tag object along with the number. Using `.text` allows us to ask just for the text that is displayed from that element.

Looking further at the results we can also notice that we have "flattened" this table to just a single column of data. This is because we are ignoring the structure of the table and just grabbing all the individual elements.

getting all files from a directory

Now that we can extract what we want from one page, we can look at extracting things from all the pages.

Yay! we have these on disk! Now we can loop over these files and start trying to get content off of them. We know we can get the content out of the html, but we first

need to get the content read in. Let's focus on that first. We are going to reuse the `target` variable, which is set to the folder with all the html files.

Path objects that are folders can use the `glob` method to do queries about their file content. We use things similar to how bash or terminal commands would work. So `*.html` will give us all the html files within that folder. (note: you can use `rglob` to recursively search a directory and all descendent child directories for those files).

This will return a `generator` object, which may look weird. But this is just a way of saving memory. You can either loop over it and print it or recast the result to a list to see all the content. The paths that are returned are already path objects!

```
for p in target.glob("*.html"):
    print(p)
```

We've got the files, and because we've used pathlib for this, these paths are already Path objects. So let's hook in bs4.

```
for p in target.glob("*.html"):
    soup = BeautifulSoup(p.read_text())
    for a in soup.select("td > p > a"):
        print(a.parent.text)
```

Basically, we take what we did before and scoot it inside our for loop. So that part isn't so complicated. What can get a bit weird is collecting everything up.

There are two big ways you could do this:

- write the contents for each page out to another file
  - extra work but ideal if there's a ton of things
  - allows you to skip writing out something you've already parsed
  - but may not be needed for smaller projects
- collect all the contents into something in memory and then write them all out
  - you'll likely want this as your end goal anyhow
  - sometimes you can just jump right to it

Writing pages out

We can use a few more pathlib tools here.

- make a new folder for these, like you have before.
- you can use `p.stem` to get just the file name from the original file, but convert it to a string
- then concat ".txt" onto it and you've made your new name!
- concat that with your new target folder and that's your new path object
- write the extracted text out to it

Something to keep in mind: the `write_text()` pathlib method doesn't work in append mode. You'll either need to collect everything up or open the file and use .write() with it.

This code takes several minutes to run! Be careful. The finished files are available to you in the repository.

```
parsed_target = pathlib.Path('parsed_sci-pages')
parsed_target.mkdir(exist_ok = True)

for p in target.glob("*.html"):
    soup = BeautifulSoup(p.read_text())
    # create the new path object
    f = parsed_target / pathlib.Path(str(p.stem) + '.txt')
    for a in soup.select("td > p > a"):
        f.write_text(a.parent.text)
```

- run it through bs first to clean up the html, you could save this content later to disk if you want (and have many of these files) as it can execute a bit slowly.
- from then you can try and use bs4 syntax to extract stuff, or you can send it through lxml.

Extracting content from the results

Given the size of these files, let's work with just one of these files!

Let's run through the results for some of this and get p tag text. This has the species plus the number of observations.

An oddity: the p tags contain each column of data. So while we can get the species names from the a tags, the structure with the text is all inside one p tag. This is a situation where we need to grab the text and then use core python to break the content apart.

```
parsed_target = pathlib.Path('parsed_sci-pages')
parsed_target.mkdir(exist_ok = True)

all_rows = []
target = pathlib.Path('sci-pages')

for p in target.glob("*Y.html"):
    soup = BeautifulSoup(p.read_text(), 'html.parser')
    f = parsed_target / pathlib.Path(str(p.stem) + '.txt')
    p_tags = soup.select("td > p")
    p_text = [p.text for p in p_tags]
    lines = []
    for chunk in p_text:
        lines.extend(chunk.split('\n'))

for l in lines:
    print(l)
```

We can also take a look at the results to see better how to filter it.

```
['',
 '',
 '',
 '',
 '',
 '',
 '',
 '',
 '',
 'Yabea microcarpa (27)',
 'Yavia cryptocarpa (7)',
 ...]
```

A quick list comprehension can help filter out all the empty strings.

```
lines = [l for l in lines if len(l) > 0]

['Yabea microcarpa (27)',
 'Yavia cryptocarpa (7)',
 'Yermo xanthocephalus (7)',
 'Youngia japonica (12)',
 'Ypsilandra thibetica (2)',
 'Yucca (2)',
 ...]
```

We could keep going with all the stuff we might want to get out of this, but let's go directly to another good example.

Using regular expressions to separate content

This is a great situation for using some regular expressions to separate out the name plus the number of species.

You can also use tools like Open Refine to help separate information like this.

Let's use the file that we made. We've got this basic pattern of `species name (number of pictures)`. We can get the species name alone just from the `a` tag, so that's sorted. But perhaps we want we get them both together to make a data file.

Connecting the link, the name, and the number could be a few more steps given that you can't get them all in a single query. But that's for another day.

You may see people out there suggesting that you use regex to snag things from html. NO NO NO NO NO. Use a proper parser to extract text content from html, and then regex if you need to separate content from that text.

Yes, this can be okay in very specific/small scale situations. Using it this way should be rare and done with extreme care if you do. I've literally never needed to do this. I once had to use string tools to fix html to make it parsable, but that's very different.

What are regular expressions?

Regex is a really cool text pattern matching system that's been around forever and usually has tools in many programs. If you've never heard the name before then you

will likely start seeing it around more.

Regex as a language is a system of metacharacters (where characters stand for something else) to describe patterns in text. There are some really advanced things you can do with this, but often most of the basics are all you'll need to get a bunch of really useful things done.

- we are used to static searching for text, where stuff needs to match verbatim even if it's a substring
- often there are systematic patterns out there that we want to search for or match
- we can describe these patterns using regex
- this allows us to just return search results matching the pattern or we can use them to actually extract out content
- many tools support regex queries, so the same sort of query can be used in python/r/other text tools. Making this sort of tool a little more "universal" if you need to use multiple platforms.

Our pattern is pretty simple, we can look at it and say: okay there are some words and text followed by some spaces and then parentheses with a number inside. Now, we haven't previewed all species names, so we should do some investigations first.

Let's read in the file and gather some information first.

```
with open('all_species.txt', 'r', encoding = 'utf-8') as infile:
    names = infile.read().splitlines() # another way of removing
newlines
```

Based on how we extracted things, we appear to have 29,814 species entries.

```
len(names)

29814
```

Are there any without parens?

```
without_p = 0

for n in names:
```

```
    if (not '(' in n) or (not ')' in n):
        without_p += 1

0
```

Apparently not. This is good! Given that these results are being generated by a database query and displayed from a template, this likely means that we successfully got all the entities that we need.

Using Regexr to help

Hands down one of my most important tools for writing regex. It can be really hard to predict stuff on your own or understand why there aren't results. https://regexr.com/ offers:

- live preview of results
- helpful tooltips about the pattern
- regex references
- save and share your patterns

Paste a sample of your data into the body of the page.

Our first goal is to write a query that will cover all the text in each line.

Remember we have three parts: some text, (), and the numbers inside them.

Some regex basics

This will be brief, but is a good preview. There are many regex learning resources plus a later workshop on this.
``

In regex we can talk about a few things:

- text in general with `[A-Za-z]` and `[0-9]` as character classes, these will match text within the ranges of those groups
  - but let's remember that there's punctuation and other accented characters
  - and that the numbers will have varying length
- directly mention a character, like `(`

    - we for sure have `(` and `)` in there, so we'll need this
- ask for things to be extracted via `()`
    - hmm this may make for a problem with the existing `()`
- indicate repetition may happen
    - so this might help with the numbers!

So we can start building this up.

- `*` for 0-infinity
    - `+` for 1-infinity
    - `{min, max}` to specify a certain number
- let's start with the numbers: `[0-9]+`
    - `[0-9]` include all digits as a character class
    - `+` operates on the previous item (not single character but item) saying that it could appear 1-infinity times.
    - This means that at least one digit must be there, but remaining flexible for what might be there.
- Specify that the numbers should be in (): `\([0-9]+\`
    - We need to add the literal versions of `(` and `)` and can do this by "escaping" them out with the `\` character.
    - Putting `\` before any single character in a regex string will have it treated as the literal version instead of the metacharacter version
- some text and stuff appears before this in the same line: `.+`
  your first inclination may be to use a character class or two to support A-Z. But remember there's punctuation, accented characters, spaces, etc. happening in here.
  We don't want to be overly specific but have the benefit of not needing to break the species names apart any more.
  `.` *stands for "anything" basically, but excludes line breaks (because these usually separate data records)*
  `+` says that something needs to appear once
  So we're matching all the content now, and need to add the `()` to the groups of data that we want to return. We only need to put it around the content that we want.

`(.+)\(([0-9]+)\)` We are ignoring the literal `()` within the query and only wanting the numbers, plus the text before it.

Alternatively: we could have attempted to "split" the text via `(` to break it into two parts.

Running queries in python

We will use the regex module, which actually has some pretty lovely official documentation. Two key functions:

- `compile` allows us to state a pattern and save it to a variable. This is good practice because this function supports a lot of flags you may want to add later on. It also lets your code stay a little more compact once you use it.
- `findall` takes a compiled pattern or a pattern directly plus text and returns all matches from your text.

```
import re

match_species = re.compile('(.+)\(([0-9]+)\)')

for n in names:
    print(re.findall(match_species, n))

[('Fabaceae sp. ', '3')]
[('Fabiana imbricata ', '5')]
[('Fabronia pusilla ', '7')]
[('Facelis retusa ', '3')]
[('Facheiroa ulei ', '2')]
...
```

We are looping over the lines and running each through this query (vs running this query on the entire thing). Thus, we get back many results. Let's collect those up.

Note that the result is a tuple inside of a list, so we'll need to extract the content. We can check the length on them to ensure that we are finding exactly one matched group. Only saving those that match and printing out any that don't.

```
match_species = re.compile('(.+)\(([0-9]+)\)')
results = []
for n in names:
    found = re.findall(match_species, n)
    if len(found) != 1:
        print(found)
    else:
        results.append(found[0])
```

Nothing appears! So great. We can also now check that we got exactly two groups from each:

```
for r in results:
    if len(r) != 2:
        print(r)
```

Nothing appears again, so we should be good.

Just another preview of a cool thing you can do, let's play with a dictionary comprehension to take these counts and make a dictionary.

```
species_pics = {name: int(count) for name, count in results}
```

Now let's convert that to a Counter object to check some details. This will convert it and then ask it to print the 10 most common (so 10 largest values).

```
from collections import Counter

counted = Counter(species_pics)

counted.most_common(10)

[('Eschscholzia californica ', 610),
 ('Unknown ', 475),
 ('Fritillaria affinis ', 393),
 ('Pinus ponderosa ', 386),
 ('Populus tremuloides ', 376),
 ('Darlingtonia californica ', 361),
 ('Ferocactus cylindraceus ', 349),
```

```
 ('Calochortus venustus ', 348),
 ('Rosa sp. ', 347),
 ('Echinocereus engelmannii ', 332)]
```

And see what the total is (which is how many pictures are on the site):

```
counted.total()

437982
```

We can also look for species with a specific number.

```
for name, count in counted.items():
    if count == 1:
        print(name)
```

Other cool tools

XPath

This system allows you to write queries to navigate and extract content from an xml tree, including html. We can use it within lxml.

Benefits:

- you can extract things with more precision than beautiful soup
- xpath has a bunch of functions etc. for precisely searching for things
- xpath is supported by many systems, including R and some other programs. so you don't need to relearn it if you migrate
- being a separate standard, lots of documentation online
- also parses regular xml
- queries run significantly faster than BeautifulSoup

Cons:

- the html needs to be correctly formed xml, but that's something that beatiful soup can do for you.
- not many people know it?

- being string queries, errors can be hard to understand

Selenium

This package focuses on interactive with websites versus just extraction. For example, pressing buttons, typing things in, clicking things, moving the page, etc.

https://www.selenium.dev/

Can be useful to automate testing or putting queries into a form. This could assist with some authentication needs, but shouldn't be used as an extraction replacement.

Python's built in `webbrowser` package

https://docs.python.org/3/library/webbrowser.html

Basically, lets you give the function a url and it will open that page up in your web browser (eg launch Chrome with that page). Usually used in conjunction with other tools.

```
import webbrowser

url = "https://www.geeksforgeeks.org"

webbrowser.open(url)
```

`pyautogui` for interface controlling

https://pyautogui.readthedocs.io/en/latest/

This lets you put in timing and actions for interacting with interfaces on your actual computer. Yes, this will actually just have your computer do stuff while you watch. It's very satisfying.

Sort of working with authentication

There are more formal ways of dealing with authentication, but when I came up against 2 factor authentication I gave up trying to use those things.

Instead, I did the following to grab a full backup of every page I could find in our internal wiki.

- I found a page that had a listing of every page within our system and manually downloaded it to disk
- Used my xpath stuff (could have used beautiful soup, too) to extract all the names and hyperlinks
- Went to the wiki and logged in with 2fa using my own credentials. I also manually went to save a page and navigated to the directory where I was saving stuff so it would come up by default later in the session.
- Looped through those urls/names, used `webbrowser` to launch the url in my browser where I was logged in.
- Used `pyautogui` to run the mac commands needed to save the page (cmd + s, wait for the file picker to pop up, press enter to save it to the default folder that shows up, close the tab)

Here's the code for reference, note that you won't have these variables set so you can't run it. It took me a solid hour of messing with this to find the right set of commands. Mostly because my laptop was set in dvorak and weird things were happening.

```
for u in urls:
    webbrowser.open(u[1])
    time.sleep(2)
    pyautogui.keyDown('command')
    pyautogui.press('s')
    pyautogui.keyUp('command')
    pyautogui.press('enter')
    time.sleep(2)
    pyautogui.keyDown('command')
    pyautogui.press('w')

    print(u)
```

Then I worked on knitting while I watched the show.