Data technologies and formats

Chris Paciorek

2025-08-26

Table of contents

Overview

References (see syllabus for links):

- Adler
- Nolan and Temple Lang, XML and Web Technologies for Data Sciences with R.
- Murrell, Introduction to Data Technologies.
- SCF tutorial: Working with large datasets in SQL, R, and Python

(Optional) Videos:

There are four videos from 2020 in the bCourses Media Gallery that you can use for reference if you want to:

1. Text files and ASCII

2. Encodings and UTF-8
3. HTML
4. XML and JSON

Note that the videos were prepared for a version of the course that used R, so there are some differences from the content in the current version of the unit that reflect translating between R and Python. I'm not sure how helpful they'll be, but they are available.

> **i** Quarto configuration details for this document
>
> This document uses the `knitr` engine and the Quarto configuration `ipynb-shell-interactivity: all`, so that output from all Python code in a chunk will print in the rendered document and the output will be interspersed with the code in the chunk rather than printed all at the end of the code chunk.

This unit focuses on interacting with information on the web, including using APIs in various ways. We'll focus on doing these manipulations in Python, but the concepts and tools involved are common to other languages, so familarity with these in Python should allow you to pick up other tools easily.

1. Working with information on the web

In this section, we'll see some ways to programmatically interact with information on the web, focusing on downloading information, but also showing how we can upload as well. As part of this we'll see some details about file formats commonly used in this context.

The main theme of this section is using tools to make it easy to interact with the web and with information in formats such as HTML, XML, JSON, and YAML.

Reading HTML

We'll cover a few basic examples in this section, but HTML and XML formatting and navigating the structure of such pages in great detail is beyond the scope of what we can cover. The key thing is to see the main concepts and know that the tools exist so that you can learn how to use them if faced with such formats.

HTML (Hypertext Markup Language) is the standard markup language used for displaying content in a web browser. In simple webpages (ignoring the more complicated pages that involve Javascript), what you see in your browser is simply a *rendering* (by the browser) of a text file containing HTML.

However, instead of rendering the HTML in a browser, we might want to use code to extract information from the HTML.

Let's see a brief example of reading in HTML tables.

Note that before doing any coding, it can be helpful to look at the raw HTML source code for a given page. We can explore the underlying HTML source in advance of writing our code by looking at the page source directly in the browser (e.g., in Firefox under the 3-lines (hamburger) "open menu" symbol, see `Web Developer (or More Tools) -> Page Source` and in Chrome `View -> Developer`

-> View Source), or by downloading the webpage and looking at it in an editor, although in some cases (such as the nytimes.com case), what we might see is a lot of JavaScript.

One lesson here is not to write a lot of your own code to do something that someone else has probably already written a package for. We'll use the BeautifulSoup4 package.

```
import io
import requests
from bs4 import BeautifulSoup as bs

URL = "https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population"

## Wikipedia requires information about the bot/program making an automated request.
user_agent = "stat243_educational_bot/0.1 (paciorek@berkeley.edu)"
headers = {'User-Agent': user_agent}
response = requests.get(URL, headers=headers)
html = response.content

# Create a BeautifulSoup object to parse the HTML
soup = bs(html, 'html.parser')

html_tables = soup.find_all('table')

## Pandas `read_html` doesn't want `str` input directly.
pd_tables = [pd.read_html(io.StringIO(str(tbl)))[0] for tbl in html_tables]

[x.shape for x in pd_tables]
```

```
[(241, 6), (13, 2), (1, 2)]
```

```
pd_tables[0]
```

```
                                Location  ...  Notes
0                                  World  ...    NaN
1                                  India  ...    [b]
2                                  China  ...    [c]
3                          United States  ...    [d]
4                              Indonesia  ...    NaN
..                                   ...  ...    ...
236         Christmas Island (Australia)  ...    NaN
237                   Niue (New Zealand)  ...    NaN
238                         Vatican City  ...   [ah]
239  Cocos (Keeling) Islands (Australia)  ...    NaN
240                Pitcairn Islands (UK)  ...    NaN

[241 rows x 6 columns]
```

Beautiful Soup works by reading in the HTML as text and then parsing it to build up a tree containing the HTML elements. Then one can search by HTML tag or attribute for information you want using

`find_all`.

As another example, it's often useful to be able to extract the hyperlinks in an HTML document.

```
URL = "http://www1.ncdc.noaa.gov/pub/data/ghcn/daily/by_year"
response = requests.get(URL)
soup = bs(response.content, 'html.parser')

## Approach 1: search for HTML 'a' tags.
a_elements = soup.find_all('a')
links1 = [x.get('href') for x in a_elements]
## Approach 2: search for 'a' elements with 'href' attribute
href_elements = soup.find_all('a', href = True)
links2 = [x.get('href') for x in href_elements]
## In either case, then use `get` to retrieve the `href` attribute value.

links2[0:9]
```

```
['..', '1763.csv.gz', '1764.csv.gz', '1765.csv.gz', '1766.csv.gz', '1767.csv.gz', '1768.csv.gz', '176
```

```
# help(bs.find_all)
```

The `kwargs` keyword arguments to `find` and `find_all` allow one to search for elements with particular characteristics, such as having a particular attribute (seen above) or having an attribute have a particular value (e.g., picking out an element with a particular `id`).

Here's another example of extracting specific components of information from a webpage (results not shown, since headlines will vary from day to day). We'll use `get_text` to retrieve the element's value.

```
URL = "https://dailycal.org"
response_news = requests.get(URL)
soup_news = bs(response_news.content, 'html.parser')
h2_elements = soup_news.find_all("h2")
headlines2 = [x.get_text() for x in h2_elements]
h3_elements = soup_news.find_all("h3")
headlines3 = [x.get_text() for x in h3_elements]
```

More generally, we may want to read an HTML document, parse it into its components (i.e., the HTML elements), and navigate through the tree structure of the HTML.

We can use CSS selectors with the `select` method for more powerful extraction capabilities. Going back to the climate data, let's extract all the `th` elements nested within `tr` elements:

```
soup.select("tr th")
```

```
[<th>Name</th>, <th>Last modified</th>, <th>Size</th>, <th>Description</th>, <th colspan="4"><hr/></t
```

Or we could extract the `a` elements whose parents are `th` elements:

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

```
[]
```

Next let's use the *XPath* language to specify elements rather than CSS selectors. XPath can also be used for navigating through XML documents.

```
import lxml.html

# Convert the BeautifulSoup object to a lxml object
lxml_doc = lxml.html.fromstring(str(soup))

# Use XPath to select elements
a_elements = lxml_doc.xpath('//a[@href]')
links = [x.get('href') for x in a_elements]
links[0:9]
```

```
['..', '1763.csv.gz', '1764.csv.gz', '1765.csv.gz', '1766.csv.gz', '1767.csv.gz', '1768.csv.gz', '176
```

XML, JSON, and YAML

XML, JSON, and YAML are three common file formats for storing data. All of them allow for key-value pairs and arrays/lists of unnamed elements and for hierarchical structure.

To read them into Python (or other languages), we want to use a package that understands the file format and can read the data into appropriate Python data structures. Usually one ends up with a set of nested (because of the hierarchical structure) lists and dictionaries.

XML

XML is a markup language used to store data in self-describing (no metadata needed) format, often with a hierarchical structure. It consists of sets of elements (also known as nodes because they generally occur in a hierarchical structure and therefore have parents, children, etc.) with tags that identify/name the elements, with some similarity to HTML. Some examples of the use of XML include serving as the underlying format for Microsoft Office and Google Docs documents and for the KML language used for spatial information in Google Earth.

Here's a brief example. The book with id attribute `bk101` is an element; the author of the book is also an element that is a child element of the book. The id attribute allows us to uniquely identify the element.

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
```

```
    <title>Midnight Rain</title>
    <genre>Fantasy</genre>
    <price>5.95</price>
    <publish_date>2000-12-16</publish_date>
   <description>A former architect battles corporate zombies, an evil sorceress, and her own ch
  </book>
</catalog>
```

We can read XML documents into Python using various packages, including `lxml` and then manipulate the resulting structured data object. Here's an example of working with lending data from the Kiva lending non-profit. You can see the XML format in a browser at http://api.kivaws.org/v1/loans/newest.xml.

XML documents have a tree structure with information at nodes. As above with HTML, one can use the *XPath* language for navigating the tree and finding and extracting information from the node(s) of interest.

Here is some example code for extracting loan info from the Kiva data. We'll first show the 'brute force' approach of working with the data as a list and then the better approach of using XPath.

```
import xmltodict

URL = "https://api.kivaws.org/v1/loans/newest.xml"
## This used to be accessible for automated download, but now gives a 403 error (access denied).
## response = requests.get(URL)
## data = xmltodict.parse(response.content)

## Instead, download to `newest.xml` file manually.
with open('newest.xml', 'r') as file:
    content = file.read()

content = content.replace("&", "and")
data = xmltodict.parse(content)

data.keys()
```

```
dict_keys(['response'])
```

```
data['response'].keys()
```

```
dict_keys(['paging', 'loans'])
```

```
data['response']['loans'].keys()
```

```
dict_keys(['@type', 'loan'])
```

```
len(data['response']['loans']['loan'])
```

```
20
```

```
data['response']['loans']['loan'][2]
```

```
{'id': '3028199', 'name': 'Delia María', 'description': {'languages': {'@type': 'list', 'language': [
1.054723 -80.45249', 'type': 'point'}}, 'partner_id': '137', 'posted_date': '2025-
08-12T17:50:17Z', 'planned_expiration_date': '2025-09-16T17:50:17Z', 'loan_amount': '1500', 'borrower
Owned Business', 'id': '6'}, {'name': '#Repeat Borrower', 'id': '28'}]}}
```

```
data['response']['loans']['loan'][2]['activity']
```

```
'Retail'
```

```
from lxml import etree
doc = etree.fromstring(content)  # formerly etree.fromstring(response.content)

loans = doc.xpath("//loan")
[loan.xpath("activity/text()") for loan in loans]
```

```
[['Poultry'], ['Retail'], ['Retail'], ['Primary/secondary school costs'], ['Farming'], ['Solar Home S
```

```
## suppose we only want the country locations of the loans (using XPath)
[loan.xpath("location/country/text()") for loan in loans]
```

```
[['Uganda'], ['Ecuador'], ['Ecuador'], ['Tajikistan'], ['Mali'], ['Honduras'], ['Pakistan'], ['Togo']
```

```
## or extract the geographic coordinates
[loan.xpath("location/geo/pairs/text()") for loan in loans]
```

```
[['-0.352537 31.552699'], ['-1.054723 -80.45249'], ['-1.054723 -80.45249'], ['39 71'], ['12.947945 -
8.863846'], ['15.136965 -87.127325'], ['31.549722 74.343611'], ['6.227277 1.581424'], ['7.955179 -
11.740995'], ['6.227277 1.581424'], ['0.008031 30.562447'], ['6.227277 1.581424'], ['6.227277 1.58142
85.544151'], ['0.481819 31.055009'], ['0.008031 30.562447'], ['-0.796014 30.596976'], ['0.481819 31.0
11.5']]
```

JSON

JSON files are structured as "attribute-value" pairs (aka "key-value" pairs), often with a hierarchical structure. Here's a brief example:

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 25,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
```

```
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [],
  "spouse": null
}
```

A set of key-value pairs is a named array and is placed inside braces (squiggly brackets). Note the nestedness of arrays within arrays (e.g., address within the overarching person array and the use of square brackets for unnamed arrays (i.e., vectors of information), as well as the use of different types: character strings, numbers, null, and (not shown) boolean/logical values. JSON and XML can be used in similar ways, but JSON is less *verbose* than XML.

We can read JSON into Python using the `json` package. Let's play again with the Kiva data. The same data that we had worked with in XML format is also available in JSON format: https://api.kivaws.org/v1/loans/newest.json.

```
URL = "https://api.kivaws.org/v1/loans/newest.json"
## This used to be accessible for automated download, but now gives a 403 error (access denied).
## response = requests.get(URL)
## Instead download manually to `newest.json`.
with open('newest.json', 'r') as file:
    content = file.read()

import json
data = json.loads(content)
type(data)
data.keys()

type(data['loans'])
data['loans'][0].keys()

data['loans'][0]['location']['country']
[loan['location']['country']  for loan in data['loans']]
```

One disadvantage of JSON is that it is not set up to deal with missing values, infinity, etc.

YAML

YAML is a similar format commonly used for configuration files that control how code/software/tools behave.

Here's an example of the YAML file specifying a GitHub Actions workflow.

Note the use of indentation (similar to Python) for nesting/hierarchy and the lack of quotation with the strings. This makes it lightweight and readable. However, the use of indentation makes it fragile (easy to have errors). Also note the use of arrays/lists and sets of key-value pairs.

```
import yaml

with open("book.yml") as stream:
   config = yaml.safe_load(stream)  ## `safe_load` avoids running embedded code.

print(config)
```

```
{'name': 'deploy-book', True: {'push': {'branches': ['main']}}, 'jobs': {'deploy-
book': {'runs-on': 'ubuntu-latest', 'steps': [{'uses': 'actions/checkout@v2'}, {'name': 'Set up Pytho
python@v1', 'with': {'python-version': 3.9}}, {'name': 'Install dependencies', 'run': 'pip install -
r book-requirements.txt\n'}, {'name': 'Build the book', 'run': 'jupyter-book build .\n'}, {'name': 'G
gh-pages@v3', 'with': {'github_token': '${{ secrets.GITHUB_TOKEN }}', 'publish_dir': './_build/html'}
```

```
## How many steps in the `deploy-book` job?
len(config['jobs']['deploy-book']['steps'])
```

```
5
```

Note that (unfortunately) `on` is treated as a boolean, as discussed in this GitHub issue for the PyYAML package.

Web APIs and webscraping

Here we'll see some examples of making requests over the Web to get data. We'll use APIs to systematically query a website for information. Ideally, but not always, the API will be documented. In many cases that simply amounts to making an HTTP GET request, which is done by constructing a URL.

The `requests` package is useful for a wide variety of such functionality. Note that much of the functionality I describe below is also possible within the shell using either `wget` or `curl`.

What is HTTP?

HTTP (hypertext transfer protocol) is a system for communicating information from a server (i.e., the website of interest) to a client (e.g., your laptop). The client sends a request and the server sends a response.

When you go to a website in a browser, your browser makes an HTTP GET request to the website. Similarly, when we did some downloading of html from webpages above, we used an HTTP GET request.

Anytime the URL you enter includes parameter information after a question mark (`www.somewebsite.com?param1=arg1&par` you are using an API.

The response to an HTTP request will include a status code, which can be interpreted based on this information.

The response will generally contain content in the form of text (e.g., HTML, XML, JSON) or raw bytes.

APIs: REST-based web services

Ideally, a web service documents their API (Application Programming Interface) that serves data or allows other interactions. REST is a popular API standard/style that we'll focus on here.

REST uses HTTP requests. When using REST, we access *resources*, which might be a Facebook account or a database of stock quotes. The API will (hopefully) document what information it expects from the user and will return the result in a standard format (often a particular file format rather than producing a webpage).

Often the format of the request is a URL (aka an endpoint) plus a query string, passed as a GET request. Let's search for plumbers near Berkeley, and we'll see the GET request, in the form:

https://www.yelp.com/search?find_desc=plumbers&find_loc=Berkeley+CA&ns=1

- the query string begins with `?`
- there are one or more `Parameter=Argument` pairs
- pairs are separated by `&`
- `+` is used in place of each space

Let's see an example of accessing economic data from the World Bank, using the documentation for their API. Following the API call structure for their "Country" API, we can download (for example), data on various countries. The documentation indicates that our REST-based query can use either a URL structure or an argument-based structure.

```
import json
## Queries based on the documentation
api_url = "https://api.worldbank.org/V2/incomeLevel/LIC/country"
api_args = "https://api.worldbank.org/V2/country?incomeLevel=LIC"

## Generalizing a bit
url = "https://api.worldbank.org/V2/country?incomeLevel=MIC&format=json"
response = requests.get(url)

data = json.loads(response.content)

## Be careful of data truncation/pagination
if False:
    url = "https://api.worldbank.org/V2/country?incomeLevel=MIC&format=json&per_page=1000"
    response = requests.get(url)
    data = json.loads(response.content)

## Programmatic control
baseURL = "https://api.worldbank.org/V2/country"
group = 'MIC'
format = 'json'
```

```
args = {'incomeLevel': group, 'format': format, 'per_page': 1000}
url = baseURL + '?' + '&'.join(['='.join(
                                [key, str(args[key])]) for key in args])
response = requests.get(url)
data = json.loads(response.content)

type(data)
```

```
<class 'list'>
```

```
len(data[1])
```

```
106
```

```
type(data[1][5])
```

```
<class 'dict'>
```

```
data[1][5]
```

```
{'id': 'BEN', 'iso2Code': 'BJ', 'name': 'Benin', 'region': {'id': 'SSF', 'iso2code': 'ZG', 'value': '
Saharan Africa '}, 'adminregion': {'id': 'SSA', 'iso2code': 'ZF', 'value': 'Sub-
Saharan Africa (excluding high income)'}, 'incomeLevel': {'id': 'LMC', 'iso2code': 'XN', 'value': 'Lo
Novo', 'longitude': '2.6323', 'latitude': '6.4779'}
```

APIs can change and disappear. A few years ago, the example above involved the World Bank's Climate Data API, which I can no longer find!

As another example, here we can see the US Treasury Department API, which allows us to construct queries for federal financial data.

In many cases you'll need to authenticate with web services that control access to the service. This can involve sending an access token with the request or going through an initial authorization procedure in your browser (e.g., logging into a Google account so that one can then interact with Google Drive via its API).

Finally, some web services allow us to pass information to the service in addition to just getting data or information. E.g., you can programmatically interact with your Facebook, Dropbox, and Google Drive accounts using REST based on HTTP POST, PUT, and DELETE requests. Authentication is of course important in these contexts and some times you would first authenticate with your login and password and receive a "token". This token would then be used in subsequent interactions in the same session.

I created your `github.berkeley.edu` accounts from Python by interacting with the GitHub API using `requests`.

HTTP requests by deconstructing an (undocumented) API

In some cases an API may not be documented or we might be lazy and not use the documentation. Instead we might deconstruct the queries a browser makes and then mimic that behavior, in some cases having to parse HTML output to get at data. Note that if the webpage changes even a little bit, our carefully constructed query syntax may fail.

Let's look at some UN data (agricultural crop data). By going to https://data.un.org/Explorer.aspx?d=FAO, and clicking on "Crops", we'll see a bunch of agricultural products with "View data" links. Click on "apricots" as an example and you'll see a "Download" button that allows you to download a CSV of the data. Let's select a range of years and then try to download "by hand". Sometimes we can right-click on the link that will download the data and directly see the URL that is being accessed and then one can deconstruct it so that you can create URLs programmatically to download the data you want.

In this case, we can't see the full URL that is being used because there's some Javascript involved. Therefore, rather than looking at the URL associated with a link we need to view the actual HTTP request sent by our browser to the server. We can do this using features of the browser (e.g., in Firefox see `Web Developer -> Network` and in Chrome `View -> Developer -> Developer tools` and choose the `Network` tab) (or right-click on the webpage and select `Inspect` and then `Network`). Based on this we can see that an HTTP GET request is being used with a URL such as: http://data.un.org/Handlers/DownloadHandler.ashx?DataFilter=itemCode:526;year:2012,2013,2014,2015,2016,2017&DataMartId=FAO&Format=csv&c=2,4,5,6,7&s=countryName:asc,elementCode:asc,year:desc.

We'e now able to easily download the data using that URL, which we can fairly easily construct using string processing in bash, Python, or R, such as this (here I just paste it together directly, but using more structured syntax such as I used for the World Bank example would be better):

Here what is returned is a zip file, which is represented in Python as a sequence of "raw" bytes, so the example code also has some syntax for handling the unzipping and extraction of the CSV file with the data.

```
import zipfile

## example URL:
## https://data.un.org/Handlers/DownloadHandler.ashx?DataFilter=itemCode:526;
##year:2012,2013,2014,2015,2016,2017&DataMartId=FAO&Format=csv&c=2,4,5,6,7&
##s=countryName:asc,elementCode:asc,year:desc
itemCode = 526
baseURL = "https://data.un.org/Handlers/DownloadHandler.ashx"
yrs = ','.join([str(yr) for yr in range(2012,2018)])
filter = f"?DataFilter=itemCode:{itemCode};year:{yrs}"
args1 = "&DataMartId=FAO&Format=csv&c=2,3,4,5,6,7&"
args2 = "s=countryName:asc,elementCode:asc,year:desc"
url = baseURL + filter + args1 + args2
## If the website provided a CSV, this would be easier, but it zips the file.
response = requests.get(url)

with io.BytesIO(response.content) as stream:  # create a file-like object
    with zipfile.ZipFile(stream, 'r') as archive:   # treat the object as a zip file
        with archive.open(archive.filelist[0].filename, 'r') as file:  # get a pointer to the embedde
            dat = pd.read_csv(file)

dat.head()
```

```
  Country or Area Element Code  ...   Value  Value Footnotes
0     Afghanistan          432  ...  202.19              NaN
1     Afghanistan          432  ...   27.45              NaN
2     Afghanistan          432  ...  134.50              NaN
3     Afghanistan          432  ...  138.05              NaN
4     Afghanistan          432  ...  138.05              NaN

[5 rows x 7 columns]
```

So, what have we achieved?

1. We have a reproducible workflow we can share with others (perhaps ourself in the future).
2. We can automate the process of downloading many such files.

Webscraping ethics and best practices

Webscraping is the process of extracting data from the web, either directly from a website or using a web API (application programming interface).

1. **Should you webscrape?** In general, if we can avoid webscraping (particularly if there is not an API) and instead directly download a data file from a website, that is greatly preferred.
2. **May you webscrape?** Before you set up any automated downloading of materials/data from the web you should make sure that what you are about to do is consistent with the rules provided by the website.

Some places to look for information on what the website allows are:

- legal pages such as Terms of Service or Terms and Conditions on the website.
- check the `robots.txt` file to see what a web crawler/automated request is allowed to do, and whether the site requires a particular delay between requests to the sites. Here are a couple examples:
  - Wikipedia
  - Google Scholar
- potentially contact the site owner if you plan to scrape a large amount of data

Here are some links with useful information:

- Blog post on webscraping ethics
- Some information on how to understand a robots.txt file

Tips for when you make automated requests:

- When debugging code that processes the result of such a request, just run the request once, save (i.e., cache) the result, and then work on the processing code applied to the result. Don't make the same request over and over again.
- In many cases you will want to include a time delay between your automated requests to a site, including if you are not actually crawling a site but just want to automate a small number of queries.

- API documentation often includes information about any limits on the rate of requests that can be made.

More details on HTTP requests

A more sophisticated way to do the download is to pass the request in a structured way with named input parameters. This request is easier to construct programmatically.

```
data = {"DataFilter": f"itemCode:{itemCode};year:{yrs}",
       "DataMartID": "FAO",
       "Format": "csv",
       "c": "2,3,4,5,6,7",
       "s": "countryName:asc,elementCode:asc,year:desc"
       }

response = requests.get(baseURL, params = data)

with io.BytesIO(response.content) as stream:
    with zipfile.ZipFile(stream, 'r') as archive:
        with archive.open(archive.filelist[0].filename, 'r') as file:
            dat = pd.read_csv(file)
```

In some cases we may need to send a lot of information as part of the URL in a GET request. If it gets to be too long (e.g,, more than 2048 characters) many web servers will reject the request. Instead we may need to use an HTTP POST request.

POST requests are also often used for submitting web forms. Here's an example POST request in which we will create a GitHub issue in an automated fashion.

```
import requests

with open(".github-access-token.txt", "r") as file:
  ghtoken = file.read().strip()

# Repository details
owner = "paciorek"
repo = "test"
url = f"https://api.github.com/repos/{owner}/{repo}/issues"

# Information about the issue in dict/json format.
issue = {
    "title": "This is an example issue",
    "body": "This is the body of the issue created via API.",
}

# Set up authentication and headers
headers = {
```

```
    "Authorization": f"token {ghtoken}",
    "Accept": "application/vnd.github+json"
}

response = requests.post(url, json=issue, headers=headers)

if response.status_code == 201:
    print(f"Successfully created Issue! {response.json()['html_url']}")
else:
    print("Could not create Issue")
    print(response.status_code, response.text)
```

Note that for security, I created a GitHub fine-grained personal access token that is limited in *scope* to only be able to handle issues in my test repository. And I've put the token into a file that is not committed to the GitHub repository containing these materials. (When doing this sort of thing with GitHub Actions, one would generally store the token as a "secret" in the repository

I could also have done this from the command line with `curl`, along the following lines:

```
curl -L \
  -X POST \
  -H "Accept: application/vnd.github+json" \
  -H "Authorization: Bearer ${GH_PERSONAL_ACCESS_TOKEN}" \
  -H "X-GitHub-Api-Version: 2022-11-28" \
  -H "User-Agent: My-Issue-Creator" \
  https://api.github.com/repos/paciorek/test/issues \
  -d '{"title":"New issue from API","body":"This issue was created using the GitHub API.","labels":["
```

where `${GH_PERSONAL_ACCESS_TOKEN}` is an environment variable containing the token.

`requests` can handle other kinds of HTTP requests such as PUT and DELETE. Finally, some websites use cookies to keep track of users, and you may need to download a cookie in the first interaction with the HTTP server and then send that cookie with later interactions. More details are available in the Nolan and Temple Lang book.

Packaged access to an API

For popular websites/data sources, a developer may have packaged up the API calls in a user-friendly fashion as functions for use from Python, R, or other software.

For example there are various Python and R packages for interacting with GitHub via its API.

Here's some example code for the `PyGitHub` package. We'll repeat the exercise of programmatically creating a GitHub issue in my `paciorek/test` repository.

```
from github import Github

with open(".github-access-token.txt", "r") as file:
  ghtoken = file.read().strip()
```

```
g = Github(ghtoken)
```

```
<string>:2: DeprecationWarning: Argument login_or_token is deprecated, please use auth=github.Auth.To
```

```
repo_name = "paciorek/test"
repo = g.get_repo(repo_name)

# Issue details
issue_title = "Test Issue Created Programmatically"
issue_body = "This is an issue filed programmatically using PyGitHub."

# Create the issue
issue = repo.create_issue(
    title=issue_title,
    body=issue_body
)

# Check the results.
print(f"Successfully created issue #{issue.number}")
```

```
Successfully created issue #25
```

```
print(f"URL: {issue.html_url}")
```

```
URL: https://github.com/paciorek/test/issues/25
```

```
g.close()
```

Accessing dynamic pages

Many websites dynamically change in reaction to the user behavior. In these cases you need a tool that can mimic the behavior of a human interacting with a site. Some options are:

- `selenium` is a popular tool for doing this, and there is a Python package of the same name.
- Using `scrapy` plus `splash` is another approach.

2. File and string encodings

Text (either in the form of a file with regular language in it or a data file with fields of character strings) will often contain characters that are not part of the limited ASCII set of characters, which has $2^7 = 128$ characters and control codes; basically what you see on a standard US keyboard. Each character takes up one byte (8 bits) of space (there is an unused bit that comes in handy in the UTF-8 context). We can actually hand-generate an ASCII file using the binary representation of each character in Python as an illustration.

The letter "M" is encoded based on the ASCII standard in bits as "01001101" as seen in the link above. For convenience, this is often written as two base-16 numbers (i.e., hexadecimal), where "0100"="4" and "1101"="d", hence we have "4d" in hexadecimal.

```
## 4d in hexadecimal is 'M'
## 0a is a newline (at least in Linux/Mac)
hexvals = b'\x4d\x6f\x6d\x0a'  # "Mom\n" in ASCII as hexadecimal

with open('tmp.txt', 'wb') as textfile:
     nbytes = textfile.write(hexvals)

nbytes
```

```
4
```

```
subprocess.run(["ls", "-l", "tmp.txt"], capture_output=True).stdout
```

```
b'-rw-r--r-- 1 paciorek scfstaff 4 Aug 21 14:57 tmp.txt\n'
```

```
with open('tmp.txt', 'r') as textfile:
     line = textfile.readlines()

line
```

```
['Mom\n']
```

When encountering non-ASCII files, in some cases you may need to deal with the text encoding (the mapping of individual characters (including tabs, returns, etc.) to a set of numeric codes). There are a variety of different encodings for text files, with different ones common on different operating systems.

We'll focus on the most common and universal approach, using Unicode as the numeric codes for characters/symbols and UTF-8 as the encoding to bytes.

Unicode includes includes more than 110,000 characters from 100 different alphabets/scripts. It's widely used on the web. One alternative that is sometimes seen is Latin-1, which encodes a small subset of Unicode and contains the characters used in many European languages (e.g., letters with accents).

Unicode characters have unique integer identifiers (the unicode *code point*), which is given by `ord` in Python. UTF-8 is the encoding that represents each Unicode character in actual bytes (in memory or on disk).

Here's an example of using non-ASCII Unicode characters. We can verify online the representation of ñ.

```
## Python `str` type stores Unicode characters.
x2_unicode = 'Pe\u00f1a 3\u00f72'
x2_unicode
```

```
'Peña 3÷2'
```

```
type(x2_unicode)
```

```
<class 'str'>
```

```
## From Unicode code point to hexadecimal representation of the code point:
ord('ñ')
```

```
241
```

```
hex(ord('ñ'))
```

```
'0xf1'
```

```
## And now to the actual UTF-8 encoding, again in hexadecimal:
bytes('\u00f1', 'utf-8')    # indeed - two bytes, not one
```

```
b'\xc3\xb1'
```

```
bytes('\u00f7', 'utf-8')    # indeed - two bytes, not one
```

```
b'\xc3\xb7'
```

```
## specified directly as hexadecimal in UTF-8 encoding
x2_utf8 = b'Pe\xc3\xb1a 3\xc3\xb72'
x2_utf8
```

```
b'Pe\xc3\xb1a 3\xc3\xb72'
```

```
with open('tmp2.txt', 'wb') as textfile:
     nbytes = textfile.write(x2_utf8)
```

Now in the shell, let's check it.

```
## Here n-tilde and division symbol take up two bytes
ls -l tmp2.txt
```

```
-rw-r--r-- 1 paciorek scfstaff 10 Aug 21 14:57 tmp2.txt
```

```
## The shell knows how to interpret the UTF-8 encoded file
## and represent the Unicode character on the screen:
cat tmp2.txt
```

```
Peña 3÷2
```

UTF-8 is cleverly designed in terms of the bit-wise representation of characters such that ASCII characters still take up one byte, and most other characters take two bytes, but some take four bytes. In fact it is even more clever than that - the representation is such that the bits of a one-byte character never appear within the representation of a two- or three- or four-byte character (and similarly for two-byte characters in three- or four-byte characters, etc.). And from the initial bit or bits, one can determine how many bytes are used for the character. For example if the first bit is a zero, it's clear that the character is an ASCII character using only one byte. If it starts with a one, then one needs to look at the next bit(s) to determine if the character takes up 2, 3, or 4 bytes.

The UNIX utility `file`, e.g. `file tmp.txt` can help provide some information.

Various Python functions such as `readlines` allow one to specify the encoding as one reads text in. The UNIX utility `iconv` and the Python function `encode` can help with conversions.

The default encoding in Python is UTF-8; note below that various types of information are interpreted in US English with the encoding UTF-8:

```
import locale
locale.getlocale()
```

```
('en_US', 'UTF-8')
```

Note that in Python and various other languages (including R and Julia), you can use Unicode characters as part of variable names:

```
peña = 7
print(peña)
```

```
7
```

```
 = 4   # \sigma = 4 (the sigma doesn't show up in the PDF version of this document)
 * peña
```

```
28
```

With strings already in Python, you can convert between encodings with the `encode` method for string objects:

```
text = 'Pe\u00f1a 3\u00f72'
text
```

```
'Peña 3÷2'
```

```
text.encode('utf-8')
```

```
b'Pe\xc3\xb1a 3\xc3\xb72'
```

```
text.encode('latin1')
```

```
b'Pe\xf1a 3\xf72'
```

```
try:
    text.encode('ascii')
except Exception as error:
    print(error)
```

```
'ascii' codec can't encode character '\xf1' in position 2: ordinal not in range(128)
```

The results above show that the two non-ASCII characters we had been working with, which required two bytes in UTF-8, require only one byte in the Latin1 (ISO 8859-1) encoding, which provides 191 characters that include ASCII characters and various characters (mostly letters with accents) used in European languages.

An error message about decoding/invalid bytes in the message often indicates an encoding issue. In particular errors may arise when trying to do read or manipulate strings in Python for which the encoding is not properly set. Here's an example with some Internet logging data that we used a few years ago in class in a problem set and which caused some problems.

```
with open('file_nonascii.txt', 'r') as textfile:
    lines = textfile.readlines()
```

```
UnicodeDecodeError: 'utf-8' codec can't decode byte 0xac in position 7922: invalid start byte
```

If we specify the file is encoded with Latin1, it works.

```
with open('file_nonascii.txt', 'r', encoding = 'latin1') as textfile:
    lines = textfile.readlines()

## Note the non-ASCII (Latin-1) character (the upside-down question mark)
lines[16925]
```

```
'from 5#c¿a7lw8lz2nX,%@ [128.32.244.179] by ncpc-email with ESMTP\n'
```