Module 5: Data Collection & Cleaning Part 1

Introduction

Collecting and cleaning data is the key topic discussed in this module. We will continue working with `pandas` and focus on the process for cleaning web data.

Our emphasis will be on extraction, transformation and loading methods. This module consists of 3 parts.

- **Part 1** - Data Sources
- **Part 2** - Web Scraping
- **Part 3** - Data Preparation

Each part is provided in a separate file. It is recommended that you follow the order of the files.

Learning Outcomes

In this module, we cover the following using Python libraries for gathering and preparing data:

- Discuss types of data
- Process data from the web
- Clean up bad data using `pandas`
- Handle missing data

Readings and Resources

The majority of the notebook draws from the recommended readings. We invite you to further supplement this notebook with the following recommended texts.

McKinney, W. (2017). Python for data analysis: Data wrangling with Pandas, NumPy, and IPython (2nd Ed.). O'Reilly Media.

- Ch. 6 (p 167-178, 186-187)
- Ch. 7 (p. 191-213)

Table of Contents

Preparing the Data: Importing and Cleaning Data

> This image shows the stages of creating a model from start to finish. (Course Authors, 2018)
> *This image shows the stages of creating a model from start to finish.*

Now that you have learned some basic functions of Python, we will begin exploring real data sources. When you clearly define your business problem, you should consider what data may be required to solve it. During this phase of the analytics methodology, you should — within the boundaries of your organizational policies — collect all data points that you think are relevant to your problem. This includes internal or external data.

Once you have the data points that may be relevant, you should conduct preliminary reviews of the data to answer some of the following questions:

- Is the data set complete?
- Is it accurate? Reliable?
- Is it biased?

- Is there enough data to draw significant conclusions about the problem?

You can address these questions by leveraging basic functions in NumPy and Pandas to organize and understand your data set.

Once you have narrowed down the data points you need, the next step is to cleanse the data such that it is consistent and usable for your analysis. In this module, we will first learn how to access data from a variety of external data sources (for example, websites), and then how to clean the data. Once the data is clean, we will be able to move on to the analysis and model development phases.

Data Sources

In recent years, many websites have made available public Application Programming Interfaces (APIs) providing data feeds (Motto, 2018). Web and Software development standards have forced a shift in the way these API's deliver data. Originally there was no standard, but with the formation of the World Wide Web Consortium (W3C) (W3C, 2018) many web technologies became standardized.

The first major shift in web development practices came with Web 2.0 (Web 2.0, nd), where web client-side technologies (i.e., what runs in your browser) were separated by concerns. What was originally only HyperText Markup Language (HTML) (WC3 Community, 2018) was split off into three main technologies:

1. **Extensible Markup Language (XML)** (W3C Community, 2016) (*content and metadata*)
2. **Style Sheets** (Bos, 2018) (*for typography or formatting*)

- The most well recognized technology being **Cascading Style Sheets (CSS)** (Mozilla and individual contributors, 2018)

3. **Client-Side JavaScript** (Hazaël-Massieux, 2016)

The second major shift came in the mid 2000's, when web technologies began standardization for Web 3.0 (*a.k.a. The Semantic Web*) (Semantic Web, nd). The main distinction between Web 3.0 and 2.0 is the separation of content (i.e. XML) from data (i.e. RDF, OWL, JSON-LD). The main technologies of Web 3.0 allow resources to be fully queryable. In other words, the shift to the semantic-web is what helped push the technology sector to become data-driven. Unfortunately, the semantic web did not pan out as once intended. The original vision of the semantic web was to decentralize data and treat the internet as a database. While the technology is in use by major companies (Schema.org community, 2018), cloud-computing and Big-Data technologies alleviated many of the problems the semantic web aimed to solve.

As of 2018, the major use of semantic web technologies is to leverage wikis to become sources of data. While this may not seem like an impressive use case, it is the necessary technology that drives IBM Watson, which was originally a centralized data source reliant entirely on querying semantic-web-enabled wikis to further increment its knowledge graph, and to intelligently query the graph to answer questions. As of 2018, the semantic-web stack and its use in IBM Watson is the most impressive example of automated *Data Cleaning* to date.

A similar trend has occurred for web applications. Web 2.0 was the standard for distinguishing web applications from regular websites. The emergence of web applications gave a new way to deliver Software-As-A-Service (SaaS) (Software as a Service, nd), with web applications now being the most popular delivery method. With the arrival of cloud-computing, SaaS became the standard for delivering information technology products. In this process, distribution of software became an increasing concern, leading to adoption of decentralized software architectures. The main pattern in use as of 2018 is that of Micro-Services (Microservices, nd) where application processes are kept as independent as possible from one another, resulting in minimal cascading failures during downtime, and easy replication and recovery of services. As such, data feeds are often given their own separate Application Programming Interface (API) (Application Programming Interface, nd) so as to separate concerns more easily (*i.e. querying for data is separated from processing data*). The consequence of this design is a large increase in network traffic over the Internet to facilitate communication between these largely independent and often replicated application processes. As of 2018, the optimal message format that reduces network traffic between application processes is JavaScript Object Notation (JSON) (Bray, 2017).

In this section, we focus on scraping data from web pages and resources, and also accessing web APIs. As data sources, web APIs have become the de-facto querying method for the majority of enterprises to both deliver services and consume data feeds.

Types of Websites

In this section we focus on extraction of data from an online data source.

Data sources can vary widely with respect to *accessibility*. Internal data sources are more easily accessible, and typically are not subject to special copyright or intellectual property laws. Internal data sources tend to be personal, or data generated by an agent or organization individually. External data sources tend to be data shared by others collectively for profiling or aggregation purposes.

Here are some examples of internal and external data:

Internal	External
Transactional	Financial markets

Internal	External
Systems health	Events and news feeds
Financial	Social media (Twitter, LinkedIn, Facebook)
Concepts / classifications	Location-based (cellphones, tracking devices)
Documents or other text	Social and economic databases
Email	Open Data
Devices	3rd-party data vendors

However, the resource types used for holding data are generally the same for both external and internal data. As such, we will focus on external data sources — the majority of which are made up of web resources. There are three variants of resources:

Web Resource Variant	Description
Static pages	Static pages refers to web resources that stay the same after being initially loaded. *(i.e. html files or any file resource such as csv, excel, pdf, etc.)*
Dynamic pages	These refer to pages with server-side or client-side code actively making changes after the initial page resources have been loaded. *(i.e. pages changed by dynamic javascript, or web application pages running server-side code.)*
APIs	These refer to interfaces for accessing data housed in applications. In recent years, these have been standardized to use JavaScript Object Notation (JSON).

In [1]:
```
# Setup code and import libraries
from bs4 import BeautifulSoup
import numpy as np
import pandas as pd
import re as re
np.random.seed(12345)
import matplotlib.pyplot as plt
plt.rc('figure', figsize=(10, 6))
np.set_printoptions(precision=4, suppress=True)
pd.options.display.max_rows = 6
```

There are two ways we can extract these resources. One is to extract the data manually (static and dynamic pages). The second is to use a provided interface (APIs, but could apply to all 3 web resource variants). We will be focusing mainly on manual extraction in this section.

First, we can use a library to access the resource and make a request.

In [2]:
```
import requests
r = requests.get('https://raw.githubusercontent.com/openmundi/world.csv/master/coun
r.text[0:140]
```

```
C:\Users\jverc\anaconda3\Lib\site-packages\urllib3\connectionpool.py:1061: InsecureR
equestWarning: Unverified HTTPS request is being made to host 'raw.githubuserconten
t.com'. Adding certificate verification is strongly advised. See: https://urllib3.re
adthedocs.io/en/1.26.x/advanced-usage.html#ssl-warnings
  warnings.warn(
```

Out[2]:
```
'Code,Name\nAF,Afghanistan\nAX,Åland Islands\nAL,Albania\nDZ,Algeria\nAS,American
Samoa\nAD,Andorra\nAO,Angola\nAI,Anguilla\nAQ,Antarctica\nAG,Antigua '
```

We then pass the data into `pandas` using a **buffer** instead of a **file descriptor**. A *buffer* is simply an array of memory and a *file descriptor* is a buffer that holds descriptive information about a file, but not the actual contents (i.e., the file's location, or where within the file a buffer is currently positioned).

In [3]:
```python
import io
# read in request as if it was a file, but is really a buffer
pd.read_csv(filepath_or_buffer=io.StringIO(r.text))
```

Out[3]:

	Code	Name
0	AF	Afghanistan
1	AX	Åland Islands
2	AL	Albania
...	...	...
246	YE	Yemen
247	ZM	Zambia
248	ZW	Zimbabwe

249 rows × 2 columns

But what if the data requested is more complicated? What if the request is to a markup file, like HTML? Or an API? Many web APIs will return a JSON string that must be loaded into a Python object. How do we convert the JSON into something usable, like a `pandas` `DataFrame`?

This is where `pandas` begins to truly shine. There are read functions in `pandas` for handling JSON and HTML. The function for handling HTML is implemented using more general libraries, meaning it actually attempts to handle XML structure first, before falling back on HTML parsing.

Both parsing methods handle multi-indexing for nested markup and return lists of type `DataFrame` for HTML with multiple tables.

In [4]:
```python
x = pd.read_html("http://www.multpl.com/s-p-500-historical-prices/table/by-month",
                 header=0,
                 skiprows=[1, 16],
                 index_col=0)
```

```
# NOTE: A list of `DataFrames` is returned
scrapedTable = x[0]
scrapedTable
```

Out[4]:

	Value
Date	
Mar 1, 2024	5115.56
Feb 1, 2024	5011.96
Jan 1, 2024	4804.49
...	...
Mar 1, 1871	4.61
Feb 1, 1871	4.50
Jan 1, 1871	4.44

1838 rows × 1 columns

In [5]:
```
scrapedTable.columns
```

Out[5]:
```
Index(['Price Value'], dtype='object')
```

But there are still restrictions on `read_html()`, such as only extracting `<table>` elements. What if another element type holds the data? Or, if the data is sporadically embedded?

Thus, we'll cover how to extract data in general from these resources. But first a closer look at how our web resource contents are structured and organized.

Web Page Contents

We begin by refreshing ourselves on the structure of each data source resource type that might be encountered and their intended use. We'll go over the most common content types and their characteristics.

HTML

Hyper Text Markup Language (HTML) (WHATWG Apple, Google, Mozilla, Microsoft, 2018) has undergone many changes since its first inception. The intent behind HTML is for **presenting** structured content. It is quite literally a language for specifying the rules for displaying content. As of 2018, HTML is ***not*** valid XML. If you held a misconception that HTML was valid XML, then be at ease. That was the goal of their respective W3C standardization committees. This is what lead to XML-valid HTML spinning-off into the X/HTML standard. However, this standard's future is uncertain due to poor adoption.

However, in general, it is simpler to assume HTML has the same structure as XML.

XML

XML consists of text, attributes, elements and special characters.

```
<rootElement>
  <childElement anAttribute="aValue">
    aTextValue
    <leafElement> anotherTextValue </leafElement>
    <emptyleafElement withAnAttribute="anotherValue" />

  </childElement>
</rootElement>
```

Unlike HTML, XML has ***no*** pre-defined elements. It is used to define other file formats. As such, XML has two ways to be evaluated.

Evaluation Type	Description
Validity	If the XML in question is well formed ***and*** also adheres to a definition or schema, like XHTML, then it is valid XML.
Well Formedness	If elements are strictly nested as a tree format and there is no ambiguity as to what is an attribute, element, and text, then the file is well-formed XML. Definitions and schemas are not required.

Validity is determined by some sort of schema or data definition. This is usually in the form of a Data Type Definition (DTD) file or an XSchema file. Both allow for grammars to be defined for the XML file, but DTD defines a context-free grammar whereas XSchema defines a context-sensitive grammar (more expressive). In order for an XML document to be well-formed, special characters are necessary. For example, how do you express a mathematical inequality without introducing a new element?

Before special characters	After special characters
`<anInEquality>1<x</anInEquality>`	`<anInEquality>1&lt;x</anInEquality>`

If you've ever wondered why `&` throw errors when editing raw XML and HTML, now you know why. It is a character reserved for denoting special characters.

NOTE: It is possible to parse malformed XML from a theoretical perspective. But, it's terribly inefficient and would slow your browser to a crawl if allowed in large one-page web applications. More precisely, XML can be parsed using a context-free grammar. HTML cannot be parsed in the same manner and requires a context-sensitive grammar, which is a more expressive grammar that requires more computation to execute its algorithm.

See Semantic Web Content for more examples.

CSS

Cascading Style Sheets (CSS) (Mozilla and individual contributors, 2018) have a simple key-value pair like format. Properties are set to values for every instance of an element selection. Element selections (a.k.a. selectors) are simply used to match on branches of HTML or XML.

```
<selector1> [, <selector2>, ...] {
  property11: <value111> [, <value112>, ...];
  ...
  property1N: <value1N1> [, <value1N2>, ...];
}
...
<selectorM1> [, <selectorM2>, ...] {
  propertyM1: <valueM11> [, <valueM12>, ...];
  ...
  propertyMN: <valueMN1> [, <valueMN2>, ...];
}
```

The following is an example snippet of CSS.

```
body {
  colour: blue;
  background-color: yellow;
  border: 1px solid black;
}
```

The snippet simply matches on a header element, similarly to how a *regular expression* might match on a substring. Upon a successful match, it does the following:

- Sets the element colour property to blue
- Sets the background colour for the displayed text to yellow
- Sets a 1 pixel thick solid (as opposed to broken or dashed) black border surrounding the element's displayed text

Afterwards, the rule continues to attempt matching on another element of the same type farther down in the document.

At this point, you might wonder why CSS exists. Isn't it doing the same thing as HTML in deciding how a page will be presented? To an extent, this is the case. However, XML with CSS is not capable of all the display options as HTML is alone. HTML actually has a superset of capabilities. However, CSS does allow for different CSS style sheets to be used for different display mediums (i.e., phone vs. tablet vs. computer vs. TV vs. projector, etc.) in a clean and modular fashion. The equivalent of this in one HTML page would be a mess of style injections for different elements, and prone to issues depending on browser support.

The overlap of CSS with HTML brings forth the primary motivation for moving away from HTML. HTML is not maintainable for complex logic in an environment where web browsers may not support standards consistently. In terms of web scraping, CSS forces a decoupling of content and style, making the it easier to scrape and clean content.

JavaScript

JavaScript is a programming language which enables interactive web content. It is often used for communicating with web applications and acts as the primary programming layer for all client-side requests. Javascript enables content and the display of pages to change during the time a webpage is viewed. Its main use is for enabling User Interface logic, but the language is a full fledged programming language and has the same general capabilities as any other programming language.

Javascript is more standardized in terms of client-side web applications. In a browser environment, Javascript makes use of the **Document Object Model (DOM)**. The DOM is a data structure that represents the current state of the web page content, and contains all information and programming event logic associated with CSS and HTML/XML (i.e., url, window dimensions, html used for rendering, HTML currently displaying, previously visited URL, etc.). To put it in simpler terms, the CSS and HTML/XML/XHTML content are all placed by the web browser into the DOM, which the browser then manipulates using Javascript. It would be accurate to say the DOM is the active state of the web page. With respect to the DOM, Javascript manipulations are forced to only trigger event listeners (like a mouse click) attached on elements, and then run a handler/function for reacting to listened events (like the effect of clicking a button).

Events are always listened to in the direction from the triggered element up to the root element. Events are handled in the reverse direction. These are event handlers.

For our purposes, we are more interested in how JavaScript is used for communicating with web applications, as this is the use case for querying data through a web API. Since requests between servers and clients are simply packets of text or binary information, Javascript can be directly injected into the content of either a request or a response. Since the browser already uses the DOM to represent all client-side data, it is often easier to use JavaScript values and objects directly to manipulate it. Hence, JavaScript Object Notation (JSON) is used to transmit message content, instead of some special messaging protocol. The primary motivation for letting the server send injectable-code (usually considered a security risk) as data is simple. The client already trusts the server, otherwise it wouldn't have connected to the URL. This is a good assumption if your web browser is its own separate process and cannot manipulate your computer's operating system.

JSON format is based on key-value pairs similar to CSS. However, rather than supporting just simple key-value pairs, JavaScript values can be arrays and objects, and not only primary values, like strings, booleans, and numbers.

```
{
  "property1": "stringValue",
  "property2": true,
  "property3": 0,
  "property4": {
    "embeddedJSONProperty": "embeddedJSONValue",
  },
  "property5": [
    {
      "arrayOfEmbeddedJSONProperty1": "value1",
    },
    {
      "arrayOfEmbeddedJSONProperty2": "value2",
    }
  ],
  "property6": [],
  "property7": null
}
```

See Semantic Web Content for more examples.

Semantic Web Content

Semantic web markup is used to describe Entities. Entities can have the following attached to them:

- Relationships
- Attributes
- Types

This is very similar to a database. However, semantic web markup is meant for describing resources so as to make data queryable. The major file formats used to describe semantic web resources are in the following table:

Format	Description	Example
Resource Description Framework (RDF)	RDF is the most minimal format. RDF is an XML format which makes full use of XSchema type values. It allows class relationships, and instance relationships to be defined along with properties and attributes.	see 1
Web Ontology Language (OWL)	OWL is more expressive than RDF. OWL is an XML format which makes full use of XSchema type values. OWL is actually Turing-complete in terms of expressiveness. Additionally, unlike RDF, OWL has an all encompassing class, a null class.	see 2
Turtle (TTL)	TTL is a non tabular record format. It is actually just a different way of writing data. TTL is not a document type like XML or HTML. In fact, it is usually defined using either RDF or OWL.	see 3
N-Triple	N-Triple is a tabular record format. It is a different way of writing data. N-Triple is not a document type like XML or HTML. In fact, it is usually defined using either RDF or OWL.	see 4

Format	Description	Example
JavaScript Object Notation for Linked Data (JSON-LD)	Since JSON has a tree structure like XML, RDF and OWL data is also expressible. The only difference is that a `type` and `id` must be specified for each resource.	see 5

Here are are some examples of these formats:

1.

~~~
<?xml
<rdf:RDF
xmlns:contact="http://www.w3.org/2000/10/swap/pim/contact#"
xmlns:eric="http://www.w3.org/People/EM/contact#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  <rdf:Description
rdf:about="http://www.w3.org/People/EM/contact#me">
    <contact:fullName>Eric Miller</contact:fullName>
  </rdf:Description>
  <rdf:Description
rdf:about="http://www.w3.org/People/EM/contact#me">
    <contact:mailbox rdf:resource="mailto:e.miller123(at)example"/>
  </rdf:Description>
  <rdf:Description
rdf:about="http://www.w3.org/People/EM/contact#me">
    <contact:personalTitle>Dr.</contact:personalTitle>
  </rdf:Description>
  <rdf:Description
rdf:about="http://www.w3.org/People/EM/contact#me">
    <rdf:type
rdf:resource="http://www.w3.org/2000/10/swap/pim/contact#Person"/>
  </rdf:Description>
</rdf:RDF>```

2.
```<rdf:RDF ...> <owl:Ontology rdf:about=""/> <owl:Class
rdf:about="#Tea"/> </rdf:RDF>```

\cite{2018a}

3.
```@prefix eric:    <http://www.w3.org/People/EM/contact#> .
@prefix contact: <http://www.w3.org/2000/10/swap/pim/contact#> .
@prefix rdf:     <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
eric:me contact:fullName "Eric Miller" .
eric:me contact:mailbox <mailto:e.miller123(at)example> .
eric:me contact:personalTitle "Dr." .
eric:me rdf:type contact:Person .```

4.
```<http://www.w3.org/People/EM/contact#me>
~~~

~~~~
<http://www.w3.org/2000/10/swap/pim/contact#fullName> "Eric Miller"
.
<http://www.w3.org/People/EM/contact#me>
<http://www.w3.org/2000/10/swap/pim/contact#mailbox>
<mailto:e.miller123(at)example> .
<http://www.w3.org/People/EM/contact#me>
<http://www.w3.org/2000/10/swap/pim/contact#personalTitle> "Dr." .
<http://www.w3.org/People/EM/contact#me>
<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
<http://www.w3.org/2000/10/swap/pim/contact#Person> .```

5.
``` {"@context": {
    "name": "http://xmlns.com/foaf/0.1/name",
    "homepage": {
      "@id": "http://xmlns.com/foaf/0.1/workplaceHomepage",
      "@type": "@id"
    },
    "Person": "http://xmlns.com/foaf/0.1/Person"
  },
  "@id": "http://me.example.com",
  "@type": "Person",
  "name": "John Smith",
  "homepage": "http://www.example.com/"}```

These numerous formats are the consequence of enabling different
data sources to be linked, regardless of format. But, the important
thing to remember is that the semantic markup can contain data, or
schema typing information, or complex value definitions. As such,
the semantic web is only as useful as the browser makes it. In the
case of semantic web sources, they are most useful when scraping
data without having to clean it.

### Images

Images generally come in two formats:

| General Formats | Description |
| --- | --- |
| Vector | This format stores all points and lines as geometric
data. As a result, scaling the image does not lead to a loss of
resolution. Instead the image is simply re-rendered. |
| Raster | This format stores all pixels individually as binary
data. As a result, the image loses resolution as it is scaled up.
However, it doesn't require a fresh computation for every rescale.
|

<img
src="https://upload.wikimedia.org/wikipedia/commons/a/aa/VectorBitmapExamp
alt="Vector vs Bitmap" style="width: 200px;"/>
 Image Source: "_File:vectorbitmapexample.svg --- wikimedia
commons, the free media repository_",  2018.  [online]
~~~~

(https://commons.wikimedia.org/w/index.php?title=File:VectorBitmapExample.svg&oldid=294435857)

In general, these images can be stored in any file format, but with the shift towards the semantic web, more vector formats tend to be XML based. The reasoning is actually related to standardization and committees spinning off from the W3C. In particular, there is a specific push by the Web3D Consortium (Web 3D Consortium, 2018) to be able to render 3D data and visualizations in a way that 3D models can be re-used by Computer Assisted Design (CAD) programs. For example, a geographic map is actually a 2D projection of a 3D model of the earth. As such, if the vector image of a map also contains all this data and renders from scratch starting with the 3D data, then this image can be reused by other geographers for other maps.

While this might seem like a rare use case, the cost to make maps using satellite raster images is quite expensive depending on the resolution. Similarly, the model of the earth is not fixed. As a result, there is a demand to be able to reuse information, even if it isn't raw data but models interpreting the data, such as 3D mathematical models of the earth.

### Micro-formats

Micro-formats are loosely defined, but in general they are formats embedded in other resources.
Let us consider Wikipedia. Many pages in Wikipedia have info-boxes via table summarizations (resource description framework, nd.). These table summarizations, though defined and rendered in HTML, are constrained by the attributes used to identify them. And this constraint is consistent across the entirety of Wikipedia, because they are actually standardized and enforced by the authors themselves. Their intended use is to provide hooks into content to extract data. As such, the micro-formats are very minimal and only require annotating class information and relationships between classes onto the format.

Example of Data	Example of Data with Micro-format
```<ul><li>Joe Doe</li><li>The Example Company</li><li>604-555-1234</li><li><a href="http://example.com/">http://example.com/</a></li></ul>```	```<ul class="vcard"><li class="fn">Joe Doe</li><li class="org">The Example Company</li><li class="tel">604-555-1234</li><li><a class="url" href="http://example.com/">http://example.com/</a></li></ul>```

The use of this information is usually for easier extraction and to link up with semantic web data. An example of this is GeoNames (GeoNames community, 2013) which extracts from multiple sources,

```
but makes use of the micro-formats for all place-taxonomy Wikipedia
pages that also have geographic coordinate systems. As a result the
data (extracted from publicly available resources) is freely
available.

As of 2018, Micro-formats have a standard being recommended as part
of the HTML standard.
```

End of Part 1

This notebook makes up one part of this module. Now that you have completed this part, please proceed to the next notebook in this module. If you have any questions, please reach out to your peers using the discussion boards. If you and your peers are unable to come to a suitable conclusion, do not hesitate to reach out to your instructor on the designated discussion board.

Links

- http://blog.miguelgrinberg.com/post/easy-web-scraping-with-python
  - About scraping using other python libraries, as well as crawling entire websites.
- http://scrapy.org/
  - About writing scrapers as configeration files via scrapy.
- https://docs.python.org/2/library/urllib2.html
  - documentation for urlib2 library
- http://docs.python-requests.org/en/latest/
- The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)
  - https://www.joelonsoftware.com/2003/10/08/the-absolute-minimum-every-software-developer-absolutely-positively-must-know-about-unicode-and-character-sets-no-excuses/
- http://import.io
  - a web-based platform for extracting data from websites without writing any code.
- http://www.crummy.com/software/BeautifulSoup/
  - popular alternative to lxml for web/screen scraping
- http://pbpython.com/web-scraping-mn-budget.html
  - Tutorial using BeautifulSoup with requests library, pandas, numpy and mathplotlib
- Python Regular Expressions Cheat Sheet
  - https://pycon2016.regex.training/cheat-sheet

References

Application programming interface, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

Bos, B. (2018). ``*Web style sheets*''. online

Bray, 2017. The javascript object notation data interchange format. onlineGeoNames community, 2013. GeoNames. online

C. S. S. working group, 2018. ``*Cascading style sheets*'', Retrieved July 2018. online

Hazaël-Massieux, D. (2016) W3C. ``*Javascript web apis - w3c*''. online

Help:infobox, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

Json-ld, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

McKinney, W. (2017). Python for data analysis, Data wrangling with pandas, numpy, and ipython (2nd edition).

Microservices, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

Motto, 2018. Todd Motto, Github - toddmotto/public-apis: a collective list of public json apis for use in web development. Retrieved July 2018. online

Mozilla and individual contributors, 2018. ``*CSS basics*''. online

Resource description framework, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

Schema.org community, (2018). ``*Schema.org*''. online

Semantic Web, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

Software as a service, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

W3C Community, 2016. X. M. L. Core Working Group, Extensible markup language (xml). online

W3C Community, 2018. W3C. Web Platform Working Group, ``*W3c html*'' online

Web 2.0, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

Web3D Consortium, 2018. Web3d consortium | open standards for real-time 3d communication. online

Web ontology language, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

WHATWG (Apple, Google, Mozilla, Microsoft), 2018. Html living standard, 2018. online

Wiki, nd. Wikipedia, the free encyclopedia, retrieved Aug 20, 2018. online

World wide web consortium (w3c), 2018. online

```
In [ ]:
```