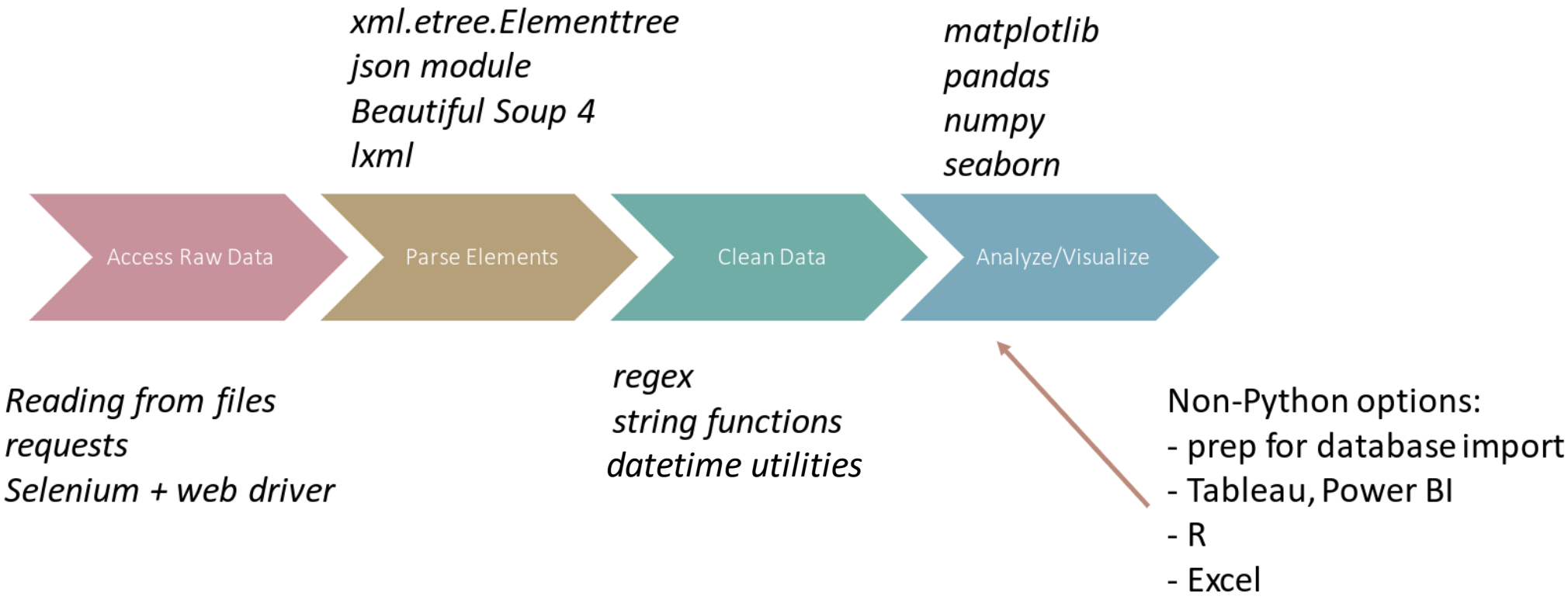
Python Workshop II:
Application Programming Interfaces (APIs)

Monica Ihli

monica@utk.edu

https://github.com/monicaihli/python-data-workshop

Data Processing Workflow



1. Get the raw data on your computer.
2. Extract into Python data structures
3. Use Python tools to clean up and restructure usable format.
4. Analyze the data

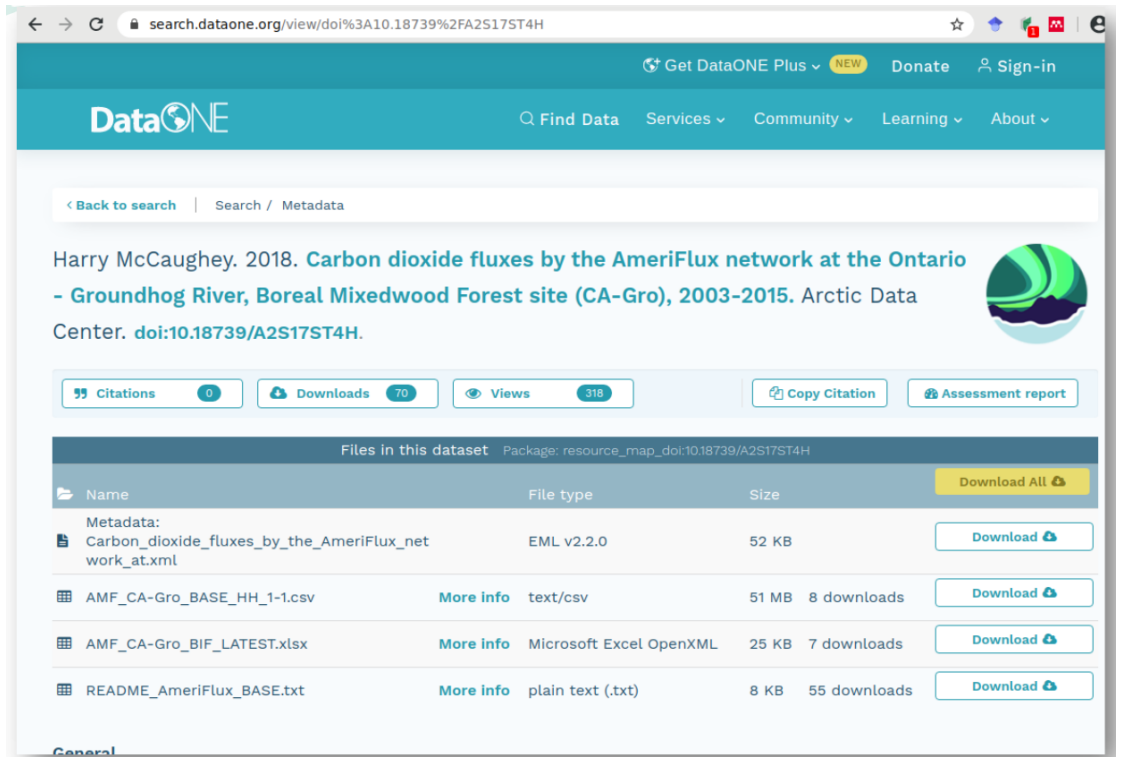
Who is This Data For?

Human-Readable Content

- Content is often navigated to (like websites).
- Presentation of content prioritizes visual appeal.
- Information is embedded in mark-up code such as HTML.
- Documents may be structured, but content is not.
- Often more work to extract and transform to orderly and native Python data types and structures.

Machine-Readable Content

- Often exposed through dedicated web service applications and locations.
- Prioritizes consistency and structure. Encoding is structural.
- Can be ingested / layered with other applications or encoding to transform raw data into aesthetically pleasing presentation. (e.g., XLST, opening data in Excel)
- Examples: text-based formats (csv, tsv), proprietary spreadsheets, XML, JSON, etc.




```
−<eml:eml packageId="doi:10.18739/A2S17ST4H" xsi:schemaLocation="https://eml.ecoinformatics.org/eml-2.2.0 eml.xsd" scope="system" sy
 −<dataset>
   <alternateIdentifier system="DOI">10.17190/AMF/1245996</alternateIdentifier>
  −<title>
     Carbon dioxide fluxes by the AmeriFlux network at the Ontario - Groundhog River, Boreal Mixedwood Forest site (CA-Gro), 2003-2015
   </title>
  −<creator>
   −<individualName>
      <givenName>Harry</givenName>
      <surName>McCaughey</surName>
    </individualName>
    <organizationName>Queen's University</organizationName>
    <electronicMailAddress>mccaughe@queensu.ca</electronicMailAddress>
    <userId directory="https://orcid.org">https://orcid.org/0000-0003-0896-1255</userId>
  </creator>
  <pubDate>2018</pubDate>
  −<abstract>
     Dynamics of carbon dioxide, water, and energy fluxes and the associated meteorological drivers; assessment of the ecological character of t
     sensing to model carbon dioxide exchange and the status of macronutrients in the canopy.
   </abstract>
  −<intellectualRights>
   −<para>
      This work is licensed under the Creative Commons Attribution 4.0 International License.To view a copy of this license, visit http://creative
    </para>
  </intellectualRights>
 −<distribution>
  −<online>
     <url>http://doi.org/doi:10.18739/A2S17ST4H</url>
   </online>
  </distribution>
 −<coverage>
  −<geographicCoverage>
   −<geographicDescription>
      Groundhog River (FCRN or CCP site 'ON-OMW') is situated in a typical boreal mixedwood forest in northeastern Ontario (48.217 degre
```

1

API – Functionality for letting two kinds of software talk to each other.

2

Web Services – An implementation of an API for transferring data over the web

3

You will often hear these terms used **interchangeably**.

Jargon for Working with APIs

API Examples

- NASA Open APIs:
  https://api.nasa.gov/
- PubMed:
  https://www.ncbi.nlm.nih.gov/home/develop/api/
- Elsevier Scopus / Science Direct/ SciVal (Scholarly literature and journal articles):
  https://dev.elsevier.com/
- Library of Congress:
  https://labs.loc.gov/lc-for-robots/
- Twitter:
  https://developer.twitter.com/en/doc

More Jargon

An API **request** will start with a **base url** (hostname + api path)

An API can have multiple **endpoints** -- the locations from which you request different resources.

Request is fleshed out with **parameters** that tell the server the specific information you are asking for.

A **rate limit** or **quota** refers to how many requests you are allowed to make in a certain amount of time.

Some services require you to aquire an **API Key** (secret passcode) to use the service.

Anatomy of a Web Services Request: Elsevier

https://api.elsevier.com/content/search/scopus?apiKey=my_api_key&
query=AF-ID(60024266) AND PUBYEAR = 2018

? separates the base url and endpoint from query parameters.

& separates multiple parameters from each other. It designates the end of one parameter and the start of another.

Anatomy of a Web Services Request: Elsevier

https://api.elsevier.com/content/search/scopus?apiKey=my_api_key&
query=AF-ID(60024266) AND PUBYEAR = 2018

Base url: https://api.elsevier.com/content/search

Endpoint: /scopus

Parameters:

apiKey=my_api_key

query=AF-ID(60024266) AND PUBYEAR = 2018

Anatomy of a Web Services Request: Pubmed OAH-PMH

https://www.ncbi.nlm.nih.gov/pmc/oai/oai.cgi?verb=ListRecords&from=2019-02-01&until=2019-10-01&set=bmcbioc&metadataPrefix=oai_dc

Base url: https://www.ncbi.nlm.nih.gov/pmc *(API I want to use)*

Endpoint: /oai/oai.cgi *(The resource communication point I'm interested in)*

Parameters:

verb=ListRecords *(From OAI-PMH standard, what I'm asking for)*

from=2019-02-01 *(Start date for interval of records I want)*

until=2019-10-01 *(End date for the records I'm interested in)*

set=bmcbioc *(Only records in this category)*

metadataPrefix=oai_dc *(The metadata format I want results in)*

Some Answers to Look for in API Documentation

Is there authentication and how does it work?

- API Key?
- IP-restricted?

What are the major end points and what are their parameters?

Is there a rate limit / quota?

How does pagination work?

- How many records per request.
- How to cycle through all records that match your query when the number of results exceeds the number that you can fetch at one time.

A Common Workflow for APIs

1. Hit endpoint with a query.
2. Get back list of identifiers which match the query.
3. Parse out the identifiers.
4. For every identifier in the list:
5. Send a new request to obtain the detailed full record for that identifier.

Python requests Library

- Third-party library, must install.
- Convenient third-party library which simplifies handling most aspects of working with **HTTP**.

Some HTTP Basics

HTTP Methods indicate the desired action. Examples:

GET – retrieve the requested resource.

POST - Submit data to the resource.

PUT – Update the target resource with the current payload.

"Payload" refers to the data sent over an HTTP request.

HTTP Headers

GENERAL headers like *Date* or *Connection* apply to request or response, but not to the content of what is being sent.

REQUEST headers like Accept, User-Agent

RESPONSE headers like *Age* or *Server* give additional context about the response.

ENTITY headers like *Content-Length, Content-Encoding* describe content of the message body.

Common Status Codes

200 – Ok *

404 – Not Found

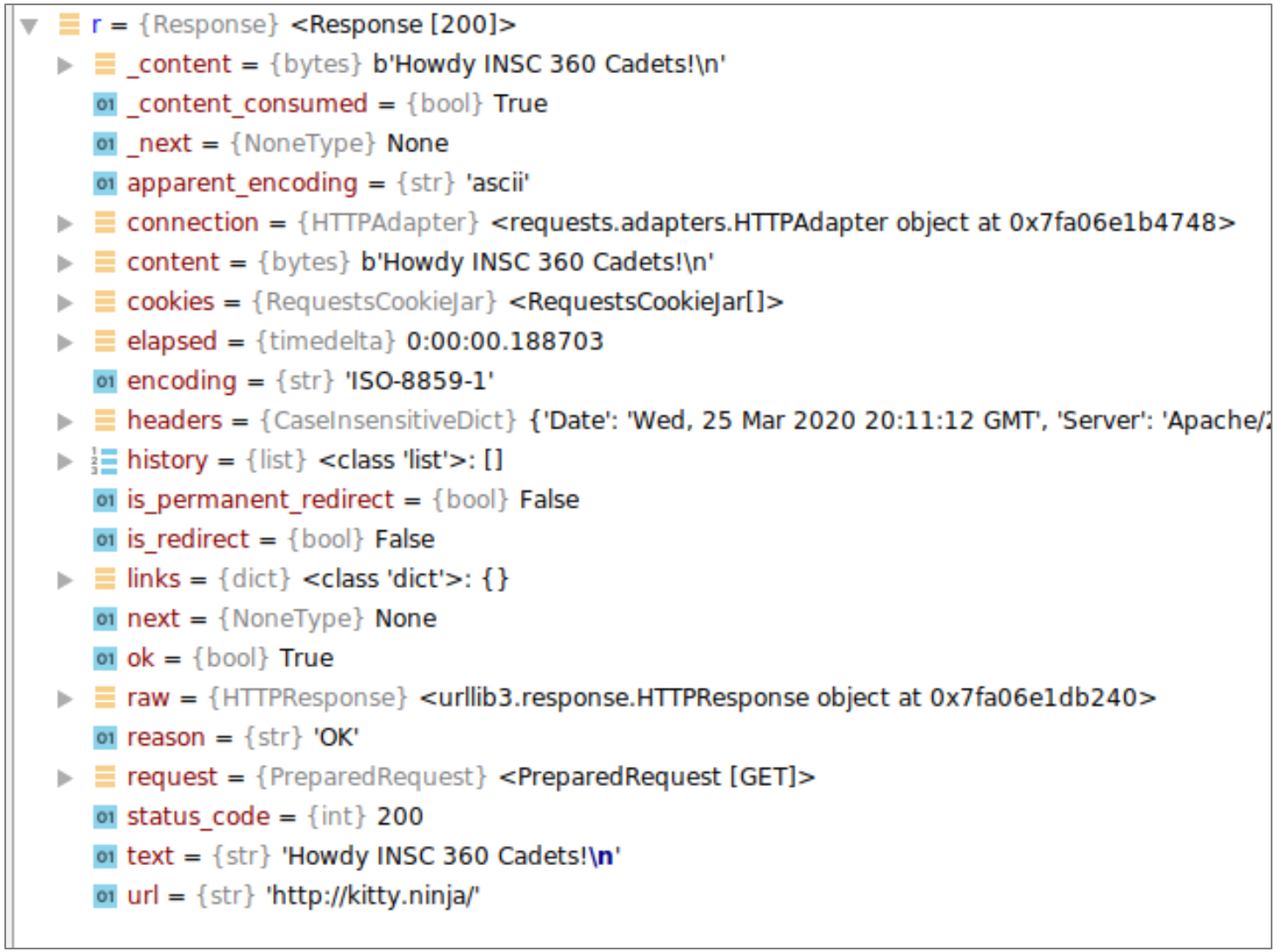
503 – Service unavailable

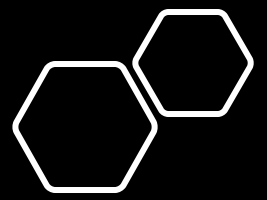
500 – Internal Server error

* means no error in http request but problems can happen on other levels!

A simple request

```
import requests
r = requests.get("http://kitty.ninja")
```





Request Arguments

url- You can provide a domain-based or even an IP address:

```
url='http://167.99.155.69')
```

params – a dict of key-value pairs representing HTTP queries and inputs:

```
payload = {'verb': 'ListRecords', 'from': '2020-03-25' }
```

headers – constructed using a dict of key-value pairs:

```
headers = {'user-agent': 'Mozilla/5.0 (X11; Ubuntu; Linux
x86_64; rv:74.0) Gecko/20100101 Firefox/74.0'} # haha look
like a human!
```

A More Complicated Request

A dictionary of key-value pairs stored in a variable called "payload", which contains search parameters.

```
r = requests.get(url=baseurl, params=payload, headers=headers)
```

Another dictionary of key-value pairs containing ordinary HTTP headers.

Web address to pull content from

Making Use of the Data

Machine Readable Formats

XML – eXtensible Markup Language

xml.etree.ElementTree

Python standard library

lxml

speedy parsing, community maintained.

beautifulSoup

mainly famous for HTML parsing but also works with an implementation of lxml. Can be more tolerant of poorly formed XML.

xml.dom.minidom

minimal DOM. Also gives us pretty XML.

xml.etree.ElementTree

This Module has two classes for representing the XML data;

- **ElementTree** – represents the whole document as a tree structure

- **Element** – represents a single node in the tree, but stores all the relationships between sub-nodes within this entity.

Data can be imported from a string (in memory) or from a file.

- .parse() - Loads XML from file into an ElementTree

- .fromstring() - Reads XML content from a string or string variable into an Element.

xml.etree.ElementTree

An element can consist of:

- **tag** – a string identifying the name of the element.
- **text** – the text contained within the start and end tag, if any.
- **tail** – in some situations, to capture additional text outside the tag.
- **attributes** – attributes are stored as a dictionary where the attribute label is the key and the value is the... value.

Strategies for Parsing XML Elements

(1) Iteratively Drilling Down – Use for loops to iterate over every child of a node, regardless of what the tag is. Repeat with additional nested for loop to drill further down the hierarchy.

(2) Drill down, one level at a time, through successive rounds of .find()/.findall() to select specific nodes which match a pattern. By default, the patterns only search immediate children.

(3) Use .find()/.findall() but provide an xpath search pattern to navigate straight to the desired node.

Searching a Tree: .find() and .findall()

- **.find()** - Finds the first child matching by tag name or path.
- **.findall()** - Finds all matching children, Returns as list.
- **.find()/.findall()** will only search immediate children unless an xpath is used: https://docs.python.org/3/library/xml.etree.elementtree.html#supported-xpath-syntax
- If no matches can be found, then .find() will return *None* type, and .findall() will return an *empty list*.

Accessing Attributes

```
<?xml version = "1.0" encoding = "UTF-8" ?>
<snacks>
    <cookies>
        <cookie sellby="03/14/2020">chocolate chip</cookie>
        <cookie sellby="03/13/2020">oatmeal raisin</cookie>
        <cookie sellby="03/15/2020">peanut butter</cookie>
    </cookies>
</snacks>
```

```
import xml.etree.ElementTree as ET
tree = ET.parse("demo1.xml")
root = tree.getroot()

list_of_cookies = root.findall(".//cookie")
for cookie in list_of_cookies:
    print(cookie.text)
    print(cookie.get("sellby"))
```

Output:

```
chocolate chip
03/14/2020
oatmeal raisin
03/13/2020
peanut butter
03/15/2020

Process finished with exit code 0
```

Searching Attributes

```
<?xml version = "1.0" encoding = "UTF-8"?>
<snacks>
    <cookies>
        <cookie sellby="03/14/2020">chocolate chip</cookie>
        <cookie sellby="03/13/2020">oatmeal raisin</cookie>
        <cookie sellby="03/15/2020">peanut butter</cookie>
    </cookies>
</snacks>
```

```
import xml.etree.ElementTree as ET
tree = ET.parse("demo1.xml")
root = tree.getroot()
# notice how the inner value of the attribute in the search is inside single quotes
this_cookie = root.find(".//cookie[@sellby='03/15/2020']")
print("Cookie batch with sell by date of 03/15/2020: " + this_cookie.text)
```

Output:

```
Cookie batch with sell by date of 03/15/2020: peanut butter

Process finished with exit code 0
```

XML vs JSON

```xml
<?xml version = "1.0" encoding = "UTF-8"?>
<snacks>
   <cookies>
      <cookie sellby="03/14/2020">chocolate chip</cookie>
      <cookie sellby="03/13/2020">oatmeal raisin</cookie>
   </cookies>
   <pies>
      <pie sellby="03/5/2020">apple</pie>
      <pie sellby="03/9/2020">pecan</pie>
   </pies>
</snacks>
```

```json
{
"snacks": [
   {
      "product": "cookie",
      "type": "chocolate chip",
      "sellby": "03/14/2020"
   },
   {
      "product": "cookie",
      "type": "oatmeal raisen",
      "sellby": "03/13/2020"
   },
   {
      "product": "pie",
      "type": "apple",
      "sellby": "03/05/2020"
   },
   {
      "product": "pie",
      "type": "pecan",
      "sellby": "03/09/2020"
   },
]}
```

JSON

Java**S**cript **O**bject **N**otation

Like XML, JSON is a structured, machine readable format which preserves hierarchical relationships between data.

Python standard library comes with json module.

JSON file extension is ".json"

Comments not supported

JSON Syntax Highlights

Data is expressed as key/value pairs.

Elements are separated by commas.

{} - Curly braces hold objects.

[] - square brackets hold arrays.

```
{
"snacks": [
   {
      "product": "cookie",
      "type": "chocolate chip",
      "sellby": "03/14/2020"
   },
   {
      "product": "cookie",
      "type": "oatmeal raisen",
      "sellby": "03/13/2020"
   },
   {
      "product": "pie",
      "type": "apple",
      "sellby": "03/05/2020"
   },
   {
      "product": "pie",
      "type": "pecan",
      "sellby": "03/09/2020"
   },
]}
```

JSON Syntax Highlights

- JSON Types:
  - String
  - Number
  - Boolean
  - Null
  - Array
  - JSON Object

- Other formats like dates must be read as strings and then use appropriate Python libraries to coerce to date/time.

Python 3 – Encoding & Decoding JSON

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

json module

- **json.load()** - reads a file containing JSON into Python object
- **json.loads()** - deserializes str, bytes, or bytesarray instance JSON content into a Python Object.
- **json.dumps()** - serializes Python data objects to a JSON formatted string.
- **json.dump()** - serializes Python obj as a JSON formatted string (supports writing to file)