Data Foundations: More File IO

Instructor: Anthony Rios

Outline

Review

More File IO

- Writing to a File
- Loading a CSV
- Creating a CSV File
- XML
- JSON

Reading the Python Documentation

Quiz

Flip the Quiz over and put down your pencil/pen when you are finished.


10 minutes

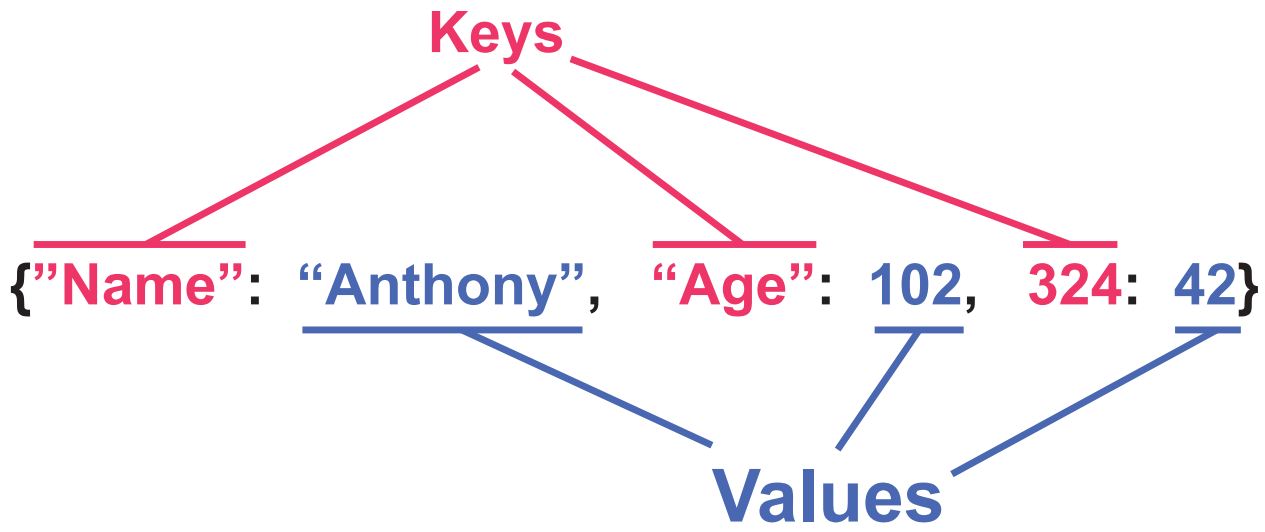
Review

More File IO

- Writing to a File
- Loading a CSV
- Creating a CSV File
- XML
- JSON

Reading the Python Documentation

Dictionaries




```
>>> myVar = {"Name": "Anthony", "Age": 102, 324: 42}
>>> myVar
{"Name": "Anthony", "Age": 102, 324: 42}
>>> myVar["Name"]
'Anthony'
>>> myVar[324]
42
```

Dictionaries: Indexing

```
>>> myVar = {"Name": "Anthony", "Age": 102, 324: 42}
```

```
>>> myVar["weight"]
File "<stdin>", line 1, in <module>
KeyError: 'weight'
```

```
>>> myVar.get("weight", 400) # myVar.get(KEY, Default Value)
 400
```

Dictionaries: Adding and Modifying New Keys/Values

```
>>> myVar = {"Name": "Anthony", "Age": 102, 324: 42} # New Dictionary

>>> myVar
{'Name': 'Anthony', 'Age': 102, 324: 42}

>>> myVar['weight'] = 400 # Create new key "weight" set value to 400

>>> myVar
{'Name': 'Anthony', 'Age': 102, 324: 42, 'weight': 400}

>>> myVar['age'] = 0 # set value of "age" to 0

>>> myVar
{'Name': 'Anthony', 'Age': 0, 324: 42, 'weight': 400}
```

Looping Over a File Line-by-Line

anthony@MacBook:~$ cat myfile.txt

```
line 1.
line 2.
line 3.
```

example.py

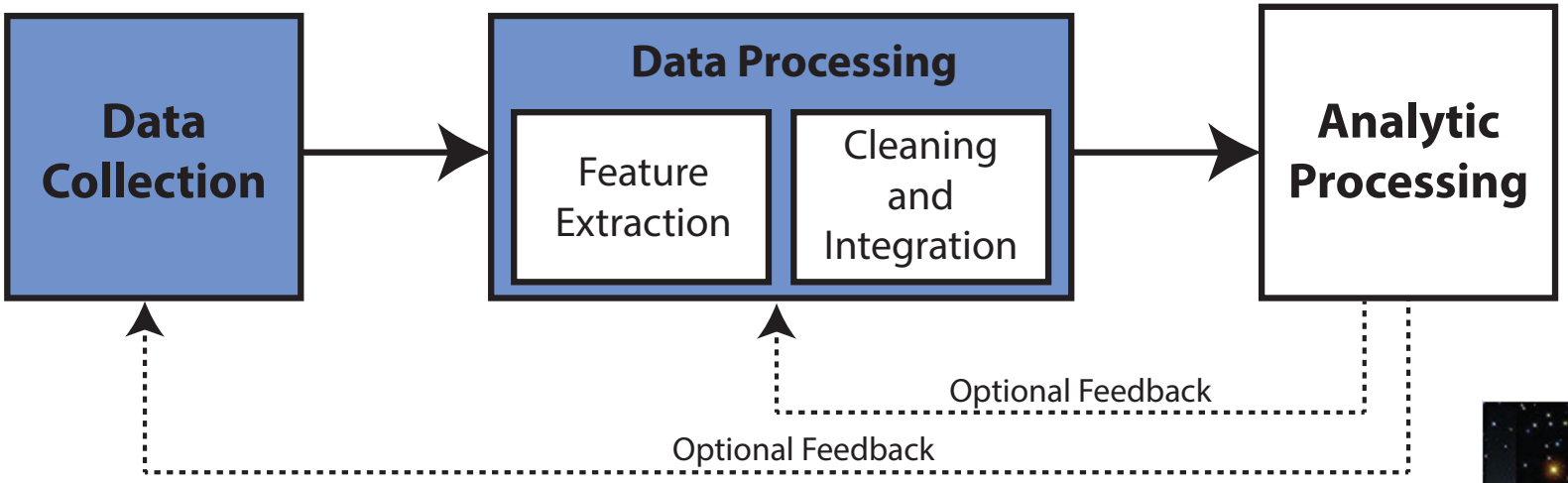
```
# Open a file with "read"
# permissions
to_open = open('myfile.txt')
# Loop over file line-by-line
for line in to_open:
        # .strip() removes white
        # space at the end and
        # the start of a string
        print(line.strip())
to_open.close() # Close the file
```

anthony@MacBook:~$ python example.py

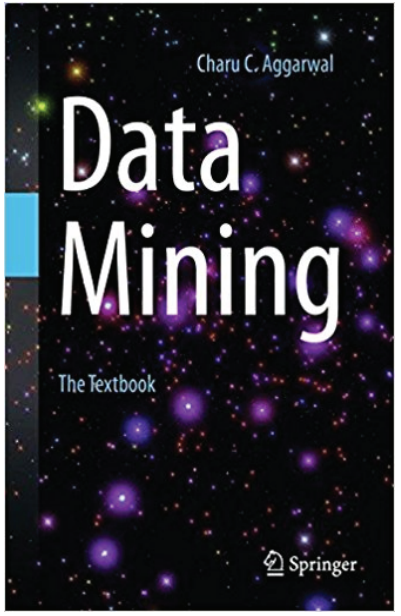
```
line 1.
line 2.
line 3.
```

Data Processing Pipeline

What we will cover in this course:



- This course focuses on data collection and data processing issues not covered in other courses.



Review

More File IO

- Writing to a File
- Loading a CSV
- Creating a CSV File
- XML
- JSON

Reading the Python Documentation

Writing to a File

myFile = open("**filename**"[, "**mode**"])

Important mode types:

- 'r' (**default**) – Read mode which is used when the file is only being read .
- 'w' – Write mode which is used to edit and write new information to the file (any existing files with the same name will be **erased** when this mode is activated) .
- 'a' – Appending mode, which is used to add new data to the **end of the file**.

Writing to a File

anthony@MacBook:~$ cat myfile.txt

```
line 1.
line 2.
line 3.
```

example.py

```
to_write = open('myfile.txt', 'w')
line = "This is a new line"
to_write.write(line)
```

anthony@MacBook:~$ cat myfile.txt

```
This is new line.
```

If "myfile.txt" does **not exist**, a new file will be **created**.

If "myfile.txt" **exists**, then the file is **overwritten**.

Appending to a File

anthony@MacBook:~$ cat myfile.txt

```
line 1.
line 2.
line 3.
```

example.py

```
to_write = open('myfile.txt', 'a')
line = "This is a new line"
to_write.write(line)
```

anthony@MacBook:~$ cat myfile.txt

```
line 1.
line 2.
line 3.This is a new line
```

Appending to a File

anthony@MacBook:~$ cat myfile.txt

```
line 1.
line 2.
line 3.
```

example.py

```
to_write = open('myfile.txt', 'a')
line = "\nThis is a new line"
to_write.write(line)
```

anthony@MacBook:~$ cat myfile.txt

```
line 1.
line 2.
line 3.
This is a new line
```

Exercise 1

Write code that reads the file “numbers.txt” line-by-line, then does the following:

- Sum all the numbers in numbers.txt, then prints the numbers to the screen.

Next, append the string ”SUM: **k**” – where **k** is the calculated sum – to the end of numbers.txt as a new line.


10 minutes

CSV Files

A CSV file (Comma Separated Values file) is a type of **plain text file** that uses specific structuring to arrange **tabular data**.

```
anthony@MacBook:~$ cat exampleCSV.csv
column 1 name,column 2 name, column 3 name
first row data 1,first row data 2,first row data 3
second row data 1,second row data 2,second row data 3
...
```

Other popular delimiters include **tab** (\t), **colon** (:), and **semi-colon** (;) characters.

CSV files are common export formats from Excel and relational databases (e.g., MySQL and MS SQL Server).

CSV Basics

```
>>> print(csv)
File "<stdin>", line 1, in <module>
NameError: name 'csv' not defined.

>>> import csv
>>> print(csv)
<module 'csv' from '/.../python3.6/csv.py'>

>>> myFile = open('myfile.csv')
>>> myCSV = csv.reader(myFile, delimiter=',')
```

Basic Format:
csv.reader(**FILE HANDLE**, delimiter=**Delimiter Character**)

Reading a CSV

anthony@MacBook:~$ cat mycsv.csv

```
name,department,birthday month
Sarah,IT,January
John,Marketing,November
```

example.py

```
import csv
myFile = open('mycsv.csv')
csvRead = csv.reader(myFile, delimiter=',')
for row in csvReader:
        print(row[1]) # prints the second column
myFile.close()
```

anthony@MacBook:~$ cat myfile.txt

```
dept.
IT
Marketing
```

Reading a CSV

anthony@MacBook:~$ cat mycsv.csv

```
name,department,birthday month
Sarah,IT,January
John,Marketing,November
```

example.py

```
csvRead = csv.reader(myFile, delimiter=',')
isHeader = True
for row in csvReader:
        if isHeader: # Ignore header
                isHeader = False
        else:
                print(row[1]) # prints the second column
```

anthony@MacBook:~$ cat myfile.txt

```
IT
Marketing
```

Exercise 2

Write code that reads the csv file "housing_prices.csv" and calculate/print the following:

- Calculate and print the sum of all house prices
- Calculate and print the average price
- Calculate and print the max price
- Print the name of the street that contains the most expensive house.


10 minutes

Creating a CSV File

example.py

```
import csv
myFile = open('new_csv.csv', 'w')
csvWriter = csv.writer(myFile, delimiter=',')
csvWriter.writerow(['col 1', 'col 2', 'col 3'])
csvWriter.writerow(['a','b','c'])
myFile.close()
```

anthony@MacBook:~$ cat new_csv.csv

```
col 1,col2,col 3
a,b,c
```

Exercise 3

Given the following list of lists

myData = [['name','department','birthday month'], ['John Doe','Marketing','November'],['Jane Smith', 'IT', 'March']]

create a csv file that is delimited with the tab '\t' character using the csv.writer() method. Name the file "employee_birthday.csv". **The list is already in the jupyter notebook under Exercise 3. Simply run the sell containing the list.**.


3 minutes

Data Example

Assume we want to send, retrieve, and display the following information:

Person

Name: Chuck
Phone (international): +1 734 303 4456
Email: Hidden

XML

- Introduced in 1996
- e**X**tensible **M**arkup **L**anguage, is a a specification for creating custom **markup languages**.
  - A **markup language** is a system for annotating a document in a way that is **syntactically distinguishable** from the text.
- XML is a **meta-language**. That means that you use it to for creating your own languages.
- The primary purpose is to help **share data** across different computers.

Person

Name: Chuck
Phone (international): +1 734 303 4456
Email: Hidden

anthony@MacBook:~$ cat myfile.xml

```
<person>
        <name>Chuck</name>
        <phone type="intl">
                +1 734 303 4456
        </phone>
        <email hide="yes" />
</person>
```

XML has **NO** predefined tags.

XML

```
anthony@MacBook:~$ cat myfile.xml
<person>
      <name>Chuck</name>
      <phone type="intl">
            +1 734 303 4456
      </phone>
      <email hide="yes" />
</person>
```

- **Start Tag** – <person>, <name>, <phone type="intl">
- **End Tag** – </person>, </name>, and </phone>
- **Content** – Chuck and +1 734 303 4456
- **Attributes** (Will be part of the start tag) – **type="intl"** and **hide="yes"**
- **Element/node** – consists of a **start** tag, **content** (optional), and an **end** tag.

XML

```
anthony@MacBook:~$ cat myfile.xml
<person>
        <name>Chuck</name>
        <phone type="intl">
                +1 734 303 4456
        </phone>
        <email hide="yes" />
</person>
```

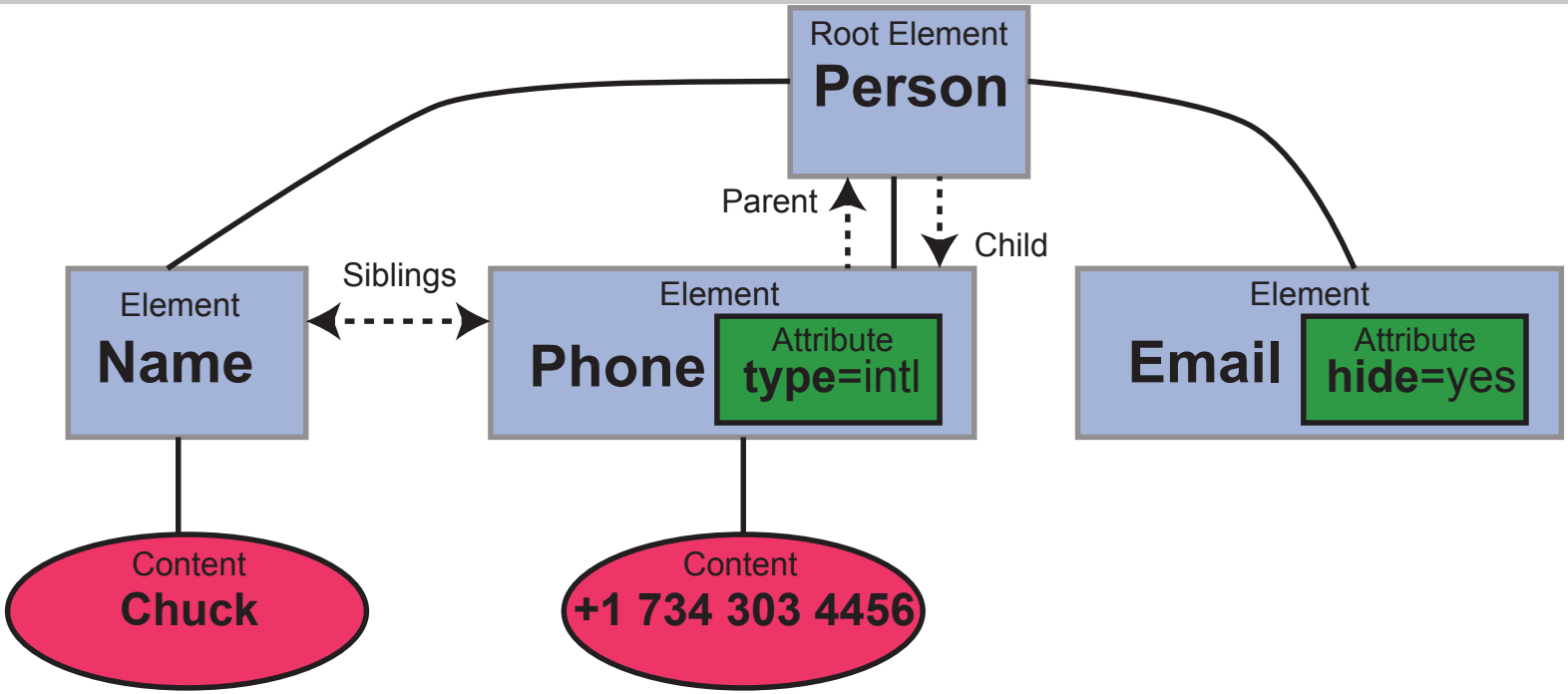
A tag is **empty** if it does not have any content <email></email>

Empty tags can also be written as <email/>

XML

anthony@MacBook:~$ cat myfile.xml

```
<person>
        <name>Chuck</name>
        <phone type="intl">
                +1 734 303 4456
        </phone>
        <email hide="yes" />
</person>
```



Parsing XML

example.py

```
import xml.etree.ElementTree as ET
data = '''<person> # ''' is used for long multi-line strings
                <name>Chuck</name>
                <phone type="intl">
                        +1 734 303 4456
                </phone>
                <email hide="yes" />
        </person>'''

tree = ET.fromstring(data) # converts a string of XML into a "tree" of nodes
# Find will return the first element that matches input parameter "name"
print('Name: {}'.format(tree.find('name').text))
# get returns a specific attribute of the element returned by find.
print('Attr: {}'.format(tree.find('email').get('hide')))
```

anthony@MacBook:~$ python example.py

```
Name: Chuck
Attr: yes
```

Looping through XML nodes/elements

anthony@MB:~$ cat myfile.xml

```
<stuff>
  <users>
    <user x="2">
      <id>001</id>
      <name>Chuck</name>
    </user>
    <user x="7">
      <id>009</id>
      <name>Brent</name>
    </user>
  </users>
</stuff>
```

anthony@MB:~$ python ex.py

```
User count: 2
Name Chuck
Id 001
Attribute 2
Name Brent
Id 009
Attribute 7
```

ex.py

```
import xml.etree.ElementTree as ET
to_open = open('myfile.xml') # Open XML file
input = to_open.read() # Read XML file into string
stuff = ET.fromstring(input)
# Returns all "user" subtrees in the XML file.
# findall takes and XPath expression as input
lst = stuff.findall('users/user')
# Count the number of elements/subtrees
# returned by findall
print('User count: {}'.format(len(lst))
for item in lst:
    print('Name {}'.format(item.find('name').text))
    print('Id {}'.format(item.find('id').text))
    print('Attribute {}'.format(item.get('x')))
```

https://www.w3schools.com/xml/xpath_syntax.asp

Exercise 4

A garden center has an XML (plant_catalog.xml) file that stores information, including price, for all plants they sell. The store is having a sale where everything is 20% off. Write a program that that prints the plant "COMMON" name, the current price, and the new sale price. An example of what the output should look like is shown below:

```
anthony@MB:~$ python ex.py
Bloodroot $2.44 to $1.95
Columbine $9.37 to $7.50
Marsh Marigold $6.81 to $5.45
...
```

Hint: You will need to use "string indexing".


10 minutes

JSON

`https://www.youtube.com/watch?v=7mj-p1Os6QA`

- First introduced in 1999
- JSON: **J**ava**S**cript **O**bject **N**otation
- JSON is a syntax for storing and exchanging data
- JSON is text, written with JavaScript Object Notation

JSON

- JSON is a **language-independent** data format

- JSON was derived from JavaScript, but as of 2017 **many programming languages** include code to generate and parse JSON-format data

JSON

Person

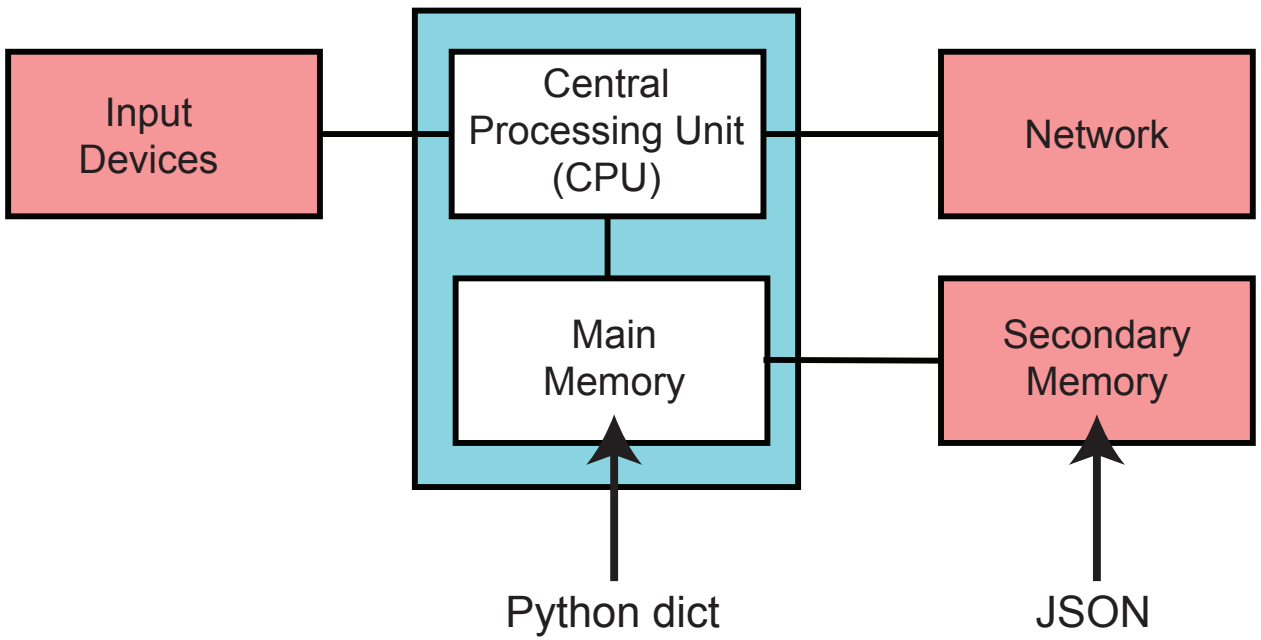
Name: Chuck
Phone (international): +1 734 303 4456
Email: Hidden

anthony@MacBook:~$ cat myfile.json

```
{
        "name" : "Chuck",
        "phone" : {
                "type" : "intl",
                "number" : "+1 734 303 4456"
        },
        "email" : {
                "hide" : "yes"
        }
}
```

JSON vs Python Dictionaries

- **JSON** is a **serialization format**. That is, JSON is a way of representing structured data in the form of a **string**.
- A **dictionary** is a **data structure**. That is, it is a way of storing data in memory that provides certain abilities to your code: in the case of dictionaries, those abilities include **rapid lookup** and **enumeration**.



JSON vs Python Dictionaries

JSON is built on **two structures**:

- A collection of **name(key)/value pairs**. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.

- An **ordered list** of values. In most languages, this is realized as an array, vector, list, or sequence.

Python's **dicts** are an implementation of one of the structures JSON is inspired by, **key/value pairs**.

JSON vs XML

JSON is Like XML Because

- Both JSON and XML are "self describing" (human readable)
- Both JSON and XML are hierarchical (values within values)
- Both JSON and XML can be parsed and used by lots of programming languages

JSON is Unlike XML Because

- JSON doesn't use end tag
- JSON is shorter
- JSON is quicker to read and write
- JSON can use arrays

JSON vs XML

Why JSON is Better Than XML

- XML is much more difficult to parse than JSON.
- JSON resembles standard key/value data structures.

For WEB applications, JSON is faster and easier than XML:

Using XML

- Fetch an XML document
- loop through the document
- Extract values and store in variables

Using JSON

- Fetch a JSON string
- Parse JSON directly into variables

Parsing JSON

>>> myJSON = '{"name":"Anthony", "age":102,
"department":"ISCS"}'

>>> myJSON['name']
Traceboack (most recent call last):
File "<stdin>", line 1
TypeError: string indices must be integers

myJSON is a **string**, not a dictionary!

Parsing JSON

```
>>> import json # Load python JSON module

>>> myJSON = '{"name":"Anthony", "age":102,
"department":"ISCS"}'

>>> JSON_to_dict = json.loads(myJSON) # Load JSON from string

>>> JSON_to_dict['name']
'Anthony'
```

Loading JSON from a File

```
anthony@MacBook:~$ cat myfile.json
[{"name":"Anthony","age":102},
        {"name":"John","age":50}]
```

```
>>> import json
>>> myFile = open('myfile.json')
>>> data = json.load(myFile)  # Return JSON from file
>>> data # JSON object stored a list of dictionaries
{'name':'Anthony','age':102},{'name':'John','age':50}
>>> data[0] # First dict. in list
{'name':'Anthony',age':102}
>>> myFile.close()
```

Convert a Dictionary to a JSON String

```
>>> data = [{'name':'Anthony','age':102},{'name':'John','age':50}]

>>> import json

>>> JSONString = json.dumps(data) # Convert data to string

>>> JSONString
'[{"name":"Anthony","age":102},{"name":"John","age":50}]'
```

Saving JSON to a File

```
>>> data = [{'name':'Anthony','age':102},{'name':'John','age':50}]

>>> import json

>>> myFile = open('myjson.json','w')

>>> json.dump(data, myFile) # Save "data" to myjson.json

>>> myFile.close()
```

Exercise 5

Using the "exampleJSON.json" file, complete the following tasks:

- Load the file into a python dictionary.
- Change the email of item with the name "Anthony" to "anthony.rios@utsa.edu"
- Add a new person to the list with the name "
- Save the new dictionary to a JSON file "exampleJSON2.json"


10 minutes

Loads vs Load and Dumps vs Dump

What is the point of Loads and Dumps?

Why would we want to convert a python dict to a string when we can save directly to a file?

- "load" will load the **entire JSON object** into memory.
  - If we have a 10GB file (or bigger), loading the entire object is **NOT** feasible.
- Similarly, "dump" requires the entire python object to be loaded in memory.
  - For streaming data, we can **NOT** store all data in memory.
  - Example: Twitter data (Imagine collecting tweets with the hashtag #DataScience for 2 months)

JSON vs JSON Lines (JSONL)

anthony@MB:~$ cat myData.**json**

```
[ {"name":"Anthony", "age":102}
{"name":"John", "age":50}
{"name":"Jane", "age":75} ]
```

anthony@MB:~$ cat myData.**jsonl**

```
{"name":"Anthony", "age":102}
{"name":"John", "age":50}
{"name":"Jane", "age":75}
```

Reading a JSONL File

anthony@MB:~$ cat myData.**jsonl**

```
{"name":"Anthony", "age":102}
{"name":"John", "age":50}
{"name":"Jane", "age":75}
```

example.py

```
import json
myFile = open('myData.jsonl')
for line in myFile: # Loop over JSON file line-by-line
    lineData = json.loads(line.strip()) # Read 1 line at a time
    print("Name: {}".format(lineData["name"]))
myFile.close()
```

anthony@MB:~$ python example.py

```
Name: Anthony
Name: John
Name: Jane
```

Exercise 6

Write code to loop over the Twitter JSONL file "twitter.jsonl" and compute the following:

- Count and print the total number of tweets (1 JSON line = 1 tweet).
- Count and print the total number of users that are in the dataset (hint: data['user']['screen_name']).
- Print the screen name of the user who has the most tweets.


15 minutes

Review

More File IO

- Writing to a File
- Loading a CSV
- Creating a CSV File
- XML
- JSON

Reading the Python Documentation

Reading the Python Documentation

Official Python Documentation:

https://docs.python.org/3/contents.html

- Contains standard API documentation.
- Has tutorials to understand syntax and built-in functions.

Case Study: Removing Items from a List

Suppose we have the following list:

>>> myList = ['john.doe@utsa.edu', 'anthony.rios@utsa.edu', 'jane.doe@utsa.edu', 'typo.email@ut3a.edu']

- We want to remove the last email because we have a typo
- We want to remove the instructors email 'anthony.rios@utsa.edu'.
- How do we do this?

`https://docs.python.org/3/contents.html`

Case Study: Remove the last item from a list

```
>>> myList = ['john.doe@utsa.edu', 'anthony.rios@utsa.edu',
'jane.doe@utsa.edu', 'typo.email@ut3a.edu']

>>> myList.pop() # Removes the last element from the list
'typo.email@ut3a.edu'

>>> myList
['john.doe@utsa.edu', 'anthony.rios@utsa.edu', 'jane.doe@utsa.edu']
```

Case Study: Remove a Specific Email from a List

```
>>> myList
['john.doe@utsa.edu', 'anthony.rios@utsa.edu', 'jane.doe@utsa.edu']

>>> myList.remove('anthony.rios@utsa.edu') # Removes the 1st occurrence of the matching object in the list

>>> myList
['john.doe@utsa.edu', 'jane.doe@utsa.edu']
```

Exercise 7

Using the file "understandingsets.txt", write code that does the following:

- Count the number of lines in the file
- Count the number of unique lines in the file (for this use a Python "set")
- Print the line that occurs the most frequently in the file (Use a Python "dict" for this).

"list" documentation:

https://docs.python.org/3/library/stdtypes.html?highlight=listlist


10 minutes

END

We have covered a lot of material – but we are going to be slowing down :)