

Working with Different File Formats



CS111 Computer Programming

Department of Computer Science
Wellesley College

How to create a dictionary from a text file?

The first 6 lines from the file `us-states.txt` in today's code folder.

```
Alabama is home to 4921532 people.  
Alaska is home to 731158 people.  
Arizona is home to 7421401 people.  
Arkansas is home to 3030522 people.  
California is home to 39368078 people.  
Colorado is home to 5807719 people.
```

Partial view of the dictionary `statesDct` created from the file `us-states.txt`.

```
In [9]: 1 statesDct
```

```
Out[9]: {'Alabama': 4921532,  
         'Alaska': 731158,  
         'Arizona': 7421401,  
         'Arkansas': 3030522,  
         'California': 39368078,  
         'Colorado': 5807719,  
         'Connecticut': 3557006,
```

Can we save this dictionary into a file, so that we can reuse it in other programs?

```
In [12]: 1 with open('statesPopulation.txt', 'w') as outFile:  
        2     outFile.write(statesDct)
```

```
-----  
TypeError                                 Traceback (most recent call last)  
<ipython-input-12-38b023f6916d> in <module>  
      1 with open('statesPopulation.txt', 'w') as outFile:  
----> 2     outFile.write(statesDct)
```

```
TypeError: write() argument must be str, not dict
```

Trying to save the dictionary into a file doesn't work immediately, because the method `write` expects a string argument.

```
In [13]: 1 with open('statesPopulation.txt', 'w') as outFile:  
        2     outFile.write(str(statesDct))
```

```
In [ ]: 1
```

Converting to a string before writing into the file does the trick, file writing is successful.

Can we read the dictionary from the file?

Partial screenshot of the file `statesPopulation.txt` that contains the dictionary, `statesDct`.

```
{'Alabama': 4921532, 'Alaska': 731158, 'Arizona': 7421401,  
'Arkansas': 3030522, 'California': 39368078, 'Colorado': 5807719,  
'Connecticut': 3557006, 'Delaware': 986809, 'Florida': 21733312,
```

We can read the content from the file, but when we do so, the result is a big string:

```
In [14]: 1 with open('statesPopulation.txt') as inputFile:  
        2     statesDct2 = inputFile.read()
```

```
In [15]: 1 type(statesDct2)
```

```
Out[15]: str
```

Since we want to work with a dictionary type, we can try to convert this string into a dictionary, as we converted the dictionary into a string.

```
In [16]: 1 statesDct3 = dict(statesDct2)
```

```
-----  
ValueError                                Traceback (most recent call last)  
<ipython-input-16-387b65aef271> in <module>  
----> 1 statesDct3 = dict(statesDct2)  
  
ValueError: dictionary update sequence element #0 has length 1; 2 is required
```

It didn't work! This is because `dict` expects at least one pair (key, value) to create a dictionary.

Takeaway:

It is not a good idea to save a dictionary into a text file (which only stores values as strings), because there is no easy way to convert a string back into a dictionary.

JSON to the rescue!

Concepts in this slide:
New module **json** with
two useful functions.

JSON (JavaScript Object Notation) is a standard format for encoding as a string (possibly nested) lists and dictionaries whose elements are numbers/strings/booleans.

```
In []: import json
```

```
In []: with open('statesPopulation.json', 'w') as outF:  
        json.dump(statesDct, outF)
```

```
In []: with open('statesPopulation.json', 'r') as inF:  
        statesDct2 = json.load(inF)
```


```
In []: statesDct2 == statesDct
```

```
Out[]: True
```

To notice:


The module **json** has two functions to deal with files: **dump** and **load**. The function **dump** is used to write data into a file, and the function **load** is used to read data from a file. They both take as an argument a file object created with the built-in function **open**.

Example: Tweets are stored in JSON format

 **Developer Platform**


Products ▾ Use cases ▾ Docs ▾ Community ▾

Updates ▾ Support Apply

Sign in 

Documentation

Search the docs

 > Twitter API

Getting started ▾

Tutorials

Tools and libraries

Migrate ▾

API reference index

The new Twitter API v2

Fundamentals ▾

Tweets ▾



Users ▾

Spaces ▾


Lists ▾




Compliance ▾

Tweet:

 **Twitter Dev** 
@TwitterDev

1/ Today we're sharing our vision for the future of the Twitter API platform!
cards.twitter.com/cards/18ce53wg...

11:24 AM · Apr 6, 2017 

 493  45  Copy link to Tweet

[Tweet your reply](#)

The following JSON illustrates the structure for these objects and some of their attributes:

```
1  {
2    "created_at": "Thu Apr 06 15:24:15 +0000 2017",
3    "id_str": "850006245121695744",
4    "text": "1/ Today we're sharing our vision for the future of the Twitter API platform!\n",
5    "user": {
6      "id": 2244994945,
7      "name": "Twitter Dev",
8      "screen_name": "TwitterDev",
9      "location": "Internet",
10     "url": "https://dev.twitter.com/",
11     "description": "Your official source for Twitter Platform news, updates & events. Need tech",
12   },
13   "place": {
14   },
15   "entities": {
16     "hashtags": [
17     ]
18   }
19 }
```

The CSV Format

(CSV = Comma Separated Values)

```
State,StatePop,Abbrev.,Capital,CapitalPop
Alabama,4921532,AL,Montgomery,198525
Alaska,731158,AK,Juneau,32113
Arizona,7421401,AZ,Phoenix,1680992
Arkansas,3030522,AR,Little Rock,197312
California,39368078,CA,Sacramento,513624
Colorado,5807719,CO,Denver,727211
```

Partial screenshot of the `us-states-more.csv` file, viewed with a text editor.

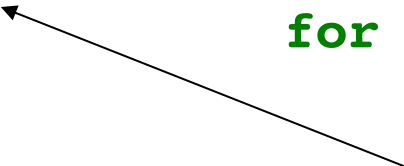
	A	B	C	D	E
1	State	StatePop	Abbrev.	Capital	CapitalPop
2	Alabama	4921532	AL	Montgomery	198525
3	Alaska	731158	AK	Juneau	32113
4	Arizona	7421401	AZ	Phoenix	1680992
5	Arkansas	3030522	AR	Little Rock	197312
6	California	39368078	CA	Sacramento	513624
7	Colorado	5807719	CO	Denver	727211

Partial screenshot of the `us-states-more.csv` file, viewed with the Google Spreadsheet editor.

CSV files are one of the most common formats to share data, since they can be displayed as a table in spreadsheet applications (Microsoft Excel, Google Spreadsheet, etc.).

Reading tuples from CSV files

```
def tuplesFromFile(filename):  
    '''Read each line from opened file,  
    strip white space,  
    split at commas,  
    convert as tuple and  
    return a list of tuples.  
    '''  
  
    with open(filename, 'r') as inputFile:  
        theTuples = [tuple(line.strip().split(','))  
                      for line in inputFile]  
  
    return theTuples
```



To notice:

We are using a list comprehension to read the content of the files into a list of tuples. This statement replaces this code:

```
theTuples = []  
for line in inputFile:  
    theTuples.append(tuple(line.strip().split(',')))
```

What happens when our data has commas?

	A	B	C	D	E
1	State	StatePop	Abbrev.	Capital	CapitalPop
2	Alabama	4921532	AL	Montgomery	198525
3	Alaska	731158	AK	Juneau	32113
4	Arizona	7421401	AZ	Phoenix	1680992
5	Arkansas	3030522	AR	Little Rock	197312
6	California	39368078	CA	Sacramento	513624
7	Colorado	5807719	CO	Denver	727211

Partial screenshot of the `us-states-more.csv` file, viewed with the Google Spreadsheet editor.

	A	B
1	Montgomery, AL	198525
2	Juneau, AK	32113
3	Phoenix, AZ	1680992
4	Little Rock, AR	197312
5	Sacramento, CA	513624
6	Denver, CO	727211
7	Hartford, CT	122105

Partial screenshot of the `capitals-only.csv` file, viewed with the Google Spreadsheet editor.

```
with open("capitals-only.csv", "w") as outF:
    for item in capitals:
        row = f"{item[0]},{item[1]}\n"
        outF.write(row)
```

```
capitals2 = tuplesFromFile("capitals-only.csv")
capitals == capitals2
```

False

Check the Notebook

It's easy to create the file about capitals from the state data, but when we read it back using the function `tuplesFromFile`, the result has tuples of three values, not two, as we desire.

The `csv` module

The `csv` module has four functions that create special objects to read/write CSV files.

<code>csv.reader</code>	creates an object that reads the content of CSV file as a list of lists
<code>csv.writer</code>	creates an object that writes a list of lists into a CSV file
<code>csv.DictReader</code>	creates an object that reads the content of CSV file as a list of dictionaries
<code>csv.DictWriter</code>	creates an object that writes a list of dictionaries into a CSV file

All the created objects are associated with a file object and can be operated upon only when the file object is open. Otherwise, we'll get an error, as shown below:

```
In [86]: import csv
with open('testFile.csv', 'w') as outputF:
    writer = csv.writer(outputF)

writer
```

```
Out[86]: <_csv.writer at 0x7ff5582a4590>
```

```
In [87]: writer.writerow(['name', 'town', 'state'])
```

```
-----
ValueError                                Traceback (most recent call last)
<ipython-input-87-50c2ee79f1fa> in <module>
----> 1 writer.writerow(['name', 'town', 'state'])
```

```
ValueError: I/O operation on closed file.
```

Examples of using the `csv` module [1]

Writing to a file using `csv.writer` and its methods `writerows()` and `writerow()`

All rows in this file will be the data about the US capitals (everything stored in the list `capitals`).

```
In [37]: with open("capitals-fixed.csv", "w") as outF:
        writer = csv.writer(outF)      # create an csv.writer object tied to the file object opened for writing
        writer.writerows(capitals)     # call method writerows to write a list into the file object
```

The first row in this file will contain the names of the columns.

```
In [44]: with open("capitals-column-names.csv", "w") as outF:
        writer = csv.writer(outF)      # create an instance of writer object
        writer.writerow(["Capital, Abbr", "Population"]) # call method writerow to write a single row
        writer.writerows(capitals)     # call method writerows to write a list into the file object
```

Reading a list of lists or a list of tuples from using `csv.reader`

Reads the content of the opened file as a list of lists.

```
In [39]: with open("capitals-fixed.csv", "r") as inF:
        reader = csv.reader(inF)      # create a csv.reader object tied to the file object opened for reading
        capitals3 = list(reader)      # the function list forces reader to iterate and read its content
        # this is similar to what we do to the range() object.
```

Reads the content of the opened file as a list of tuples, using list comprehension.

```
In [42]: with open("capitals-fixed.csv", "r") as inF:
        reader = csv.reader(inF)
        capitals4 = [tuple(row) for row in reader] # read each row and convert it to a tuple
```

Examples of using the `csv` module [2]

Reading from a file using `csv.DictReader`

This code will read from a CSV file into a list of dictionaries, each one corresponding to a row.

```
In [47]: with open("capitals-column-names.csv", "r") as inF:
         dctReader = csv.DictReader(inF)      # create an object of the csv.DictReader tied to the file object
         dctRows = list(dctReader)           # read the content of dctReader, forcing it to iterate
```

Each element of a list is a dictionary of a special kind, known in Python as an `OrderedDict`.

```
In [49]: oneRow = dctRows[0]
         oneRow
```

```
Out[49]: OrderedDict([('Capital', 'Montgomery', 'AL'), ('Population', '198525')])
```

Writing a list of dictionary using `csv.DictWriter`

We can rearrange the order of columns and pass them as an argument to `csv.DictWriter`. Notice that `csv.DictWriter` has a special method to write the header (first row) of a CSV file.

```
In [55]: columns = ['Abbrev.', 'State', 'Capital', 'CapitalPop', 'StatePop']
```

```
In [56]: with open("us-states-rearranged.csv", "w") as outF:
         writer = csv.DictWriter(outF, fieldnames=columns)
         writer.writeheader()      # this method doesn't need an argument, it uses the fieldnames parameter
         writer.writerows(statesList) # writes the list of dictionaries into the file object
```

Test your knowledge

1. What do the acronyms JSON and CSV stand for?
2. In what ways do these two formats differ from one another?
3. Which format allows programmers more flexibility in transferring data? Why?
4. What do the two functions `dump` and `load` of the **json** module do?
5. What is the difference between `csv.reader` and `csv.DictReader`?
6. What is the difference between `csv.writer` and `csv.DictWriter`?
7. When might it be useful to use the `csv.writer` method `writerow`?
8. What is the role of `list` in the expression: `list(reader)` [Slide 10, cell 39]
9. What does the method `writeheader` do?
10. What is the advantage of using `csv.DictWriter` over `csv.writer`?