Programming fundamentals with Python
Using files with Python

Pepe García jgarciah@faculty.ie.edu

Plan for today

- Files refresher

Plan for today

- Files refresher
- CSV refresher

Plan for today

- Files refresher
- CSV refresher
- Learn about JSON

data for today

Session 6 repository

All materials for today's session will be in
`https://github.com/pfp-2021/session-6`. Clone it if you want to have it
in your computer.

Plan for today

- Files refresher

the **open** function

We can use **open()** to open a file in Python, we only need to pass the path of the file we want to open. Let's say there's a file named hello.txt in my desktop that I want to open and read from Python, I can do it as follows:

the **open** function

We can use **open()** to open a file in Python, we only need to pass the path of the file we want to open. Let's say there's a file named hello.txt in my desktop that I want to open and read from Python, I can do it as follows:

```python
file = open("/Users/pepe/Desktop/hello.txt")
```

Reading the contents of a file

Now that we know how to open and close files, we can read the contents of a file. Let's do that line by line.

Reading the contents of a file

Now that we know how to open and close files, we can read the
contents of a file. Let's do that line by line.

```python
file = open("/Users/pepe/Desktop/hello.txt")

for line in file:
    print(line)

file.close()
```

Reading the contents of a file

Now that we know how to open and close files, we can read the contents of a file. Let's do that line by line.

```python
file = open("/Users/pepe/Desktop/hello.txt")

for line in file:
    print(line)

file.close()
```

As you can see, we're treating `file` as a list of lines.

there is a useful Python keyword that one can use to make sure that the file will always be closed, **with**:

there is a useful Python keyword that one can use to make sure that the file will always be closed, **with**:

```python
with open("file_path") as file:

    for line in file:
        #do something with line
        print(line)
```

Handling files. modes

When opening a file, we can choose in which **mode** we open it depending on how we're going to use it.

I/O Mode	Syntax	Behavior
Read	'r'	Opens the contents of a file for reading into the file interface, allowing for lines to be read-in successively.
Write	'w'	Creates a file with the specified name and allows for text to be written to the file; note that specifying a pre-existing filename will overwrite the existing file.
Append	'a'	Opens an existing file and allows for text to be written to it, starting at the conclusion of the original file contents.
Read and Write	'r+'	Opens a file such that its contents can be both read-in and written-to, thus offering great versatility.

Python's available file-access modes are summarized here.

Figure 1: file modes

Writing files

We can write into files in a way similar to the one used for reading them.

Writing files

We can write into files in a way similar to the one used for reading them.

```python
with open('/Users/pepe/Desktop/goodbye.txt', 'w') as file:
    file.write("goodbye y'all!")
```
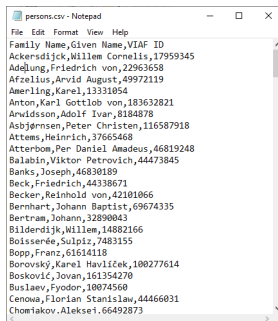
Plan for today

- Files refresher
- CSV refresher

CSV

CSV is a data interchange format used for representing tabular data.

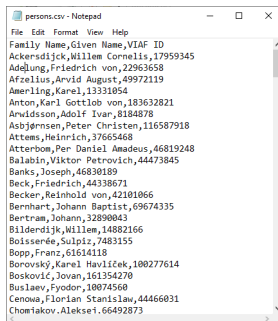# CSV - how does it look like?

CSV - how does it look like?

- **syntax** is, just the values separated by commas

CSV - how does it look like?

- **syntax** is, just the values separated by commas
- We separate entries by adding a new line

CSV files - reading

The **csv** library is based on the idea of readers and writers. One can read all lines in a file like so:

```python
import csv

with open("file.csv") as f:
    reader = csv.reader(f)
    for line in reader:
        print(line)  #line is a list
```

CSV files - reading

The **csv** library is based on the idea of readers and writers. One can read all lines in a file like so:

```python
import csv

with open("file.csv") as f:
    reader = csv.reader(f)
    for line in reader:
        print(line)  #line is a list
```

- first we open the file normally

CSV files - reading

The **csv** library is based on the idea of readers and writers. One can read all lines in a file like so:

```python
import csv

with open("file.csv") as f:
    reader = csv.reader(f)
    for line in reader:
        print(line)  #line is a list
```

- first we open the file normally
- Then we create a reader using **csv.reader()**

CSV files - reading

The **csv** library is based on the idea of readers and writers. One can read all lines in a file like so:

```python
import csv

with open("file.csv") as f:
    reader = csv.reader(f)
    for line in reader:
        print(line)  #line is a list
```

- first we open the file normally
- Then we create a reader using **csv.reader()**
- Finally, we operate with the reader

CSV files - writing

writing is not very different from reading:

```python
lines = [
  ["asdf", "qwer"],
  ["hello", "world"]
]

with open("file.csv", "a") as f:
    writer = csv.writer(f)
    for line in lines:
        writer.writerow(line)
```

CSV Files - exercise

Let's remember how to use CSV files. There is a CSV in
`data/data.csv`. Let's create a Python function that returns all the
emails from the users in the file.

Plan for today

- Files refresher
- CSV refresher
- Learn about JSON

JSON

JSON (http://json.org) is a data
interchange format, like CSV.
The name JSON stands for
Javascript Object Notation,
because the way of writing it is
very similar to Javascript.
The main difference is that
**JSON can represent arbitrary
data, not only tabular data**.
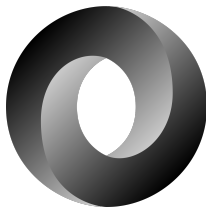
JSON - how does it look like?

```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

JSON - how does it look like?

```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

- **syntax** similar to Python data structures

JSON - how does it look like?

```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

- **syntax** similar to Python data structures
- supports **primitive** datatypes (**int, str, bool, float**).

JSON - how does it look like?

```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

- **syntax** similar to Python data structures
- supports **primitive** datatypes (**int, str, bool, float**).
- supports collections of elements with **lists**

JSON - how does it look like?

```
{
    "orders": [
        {
            "orderno": "748745375",
            "date": "June 30, 2088 1:54:23 AM",
            "trackingno": "TN0039291",
            "custid": "11045",
            "customer": [
                {
                    "custid": "11045",
                    "fname": "Sue",
                    "lname": "Hatfield",
                    "address": "1409 Silver Street",
                    "city": "Ashland",
                    "state": "NE",
                    "zip": "68003"
                }
            ]
        }
    ]
}
```

- **syntax** similar to Python data structures
- supports **primitive** datatypes (**int, str, bool, float**).
- supports collections of elements with **lists**
- supports mapping of elements with **dictionaries**

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

- lists

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

- lists
- integers

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

- lists
- integers
- booleans

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

- lists
- integers
- booleans
- strings

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

- lists
- integers
- booleans
- strings
- floats

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

- lists
- integers
- booleans
- strings
- floats
- null (an empty value)

JSON

JSON can contain

```
[1, 2, 3]
1
true
"potatoes"
4.77
null
{"name":"Pepe","surname":"Garcia"}
```

- lists
- integers
- booleans
- strings
- floats
- null (an empty value)
- dictionaries

JSON

JSON is very similar to how we declare our data in Python but the cool thing about it is that it can be used **from any language**. In Python we will be able to use JSON using the `json` module

```
import json
```

JSON - reading JSON data in Python

As with other formats we've seen so far, in order to operate with json files we will first **open()** the file.

JSON - reading JSON data in Python

As with other formats we've seen so far, in order to operate with json files we will first **open()** the file.

```python
import json

with open("data.json") as file:
    json_data = json.load(file)

    for key in json_data:
        print(key)
```

json.load is a function from the json module that takes a **file object** as parameter and returns the contents of that file **parsed as JSON**.

JSON - writing JSON files

The process of writing JSON files is similar to what we know already.

```python
import json

data = {
  "name": "Pepe",
  "last_name": "Garcia"
}

with open("data.json", "w") as file:
    json.dump(data, file)
```

As you can see, we're calling `json.dump` from the `json` library, and passing first the data we want to write to the file and then the file object as parameters.

Homework

You will find the data files for these exercises in this repository:
https://github.com/pfp-2021/session-6

- Let's get personal data from the person represented in `luke.json`. Print the `name`, `height`, `eye_color`, and `mass`.
- Let's create a **format conversor**. Our function `convert_format` will read all the data from `data/data.csv` and write it to a new **JSON** file named `converted.json`