Lesson Overview

You already know, how to work with text formats. Therefore, it is the right time for **File formats**.

What lies ahead of you in this lesson?

- Working with **CSV files** (table formats), which can be opened in MS Excel.
- Proper usage of **JSON files** which are used for data transfer amongst various web apps.

Both are very common file formats therefore it is important to have a certain level of undertanding here. Of course, you'll get the very basics so you can start working with them, but from here onwards you should have a solid base ;)

Create Files

PYTHON ACADEMY / 11. FILE FORMATS / REVIEW EXERCISES / CREATE FILES

First, create a file called `'ToBe_Created.txt'`. The file should contain the names of files to be created:

```
report_2017.xlsx
report_2017.xls

summary_2017.pptx
outlook_2018.pptx
template overtime increase.docx
```

```
icon_12123.jpg
smile.gif
paths.png
workflow_improvement.png
testing_process.pdf
report_2016.pdf
```

Next, you can create **your script** that will:

1. Create a directory called `TestDir`
2. Read the file `'ToBe_Created.txt'`
3. Create new files inside the `TestDir` folder and name them using names listed in `'ToBe_Created.txt'` file.

The script should be launched from the command line. Script should expect 2 command line arguments:

1. Absolute path to the directory `TestDir` that you would like to create
2. Absolute path to the file `ToBe_Created.txt`

Example of use:

```
$ python create_file.py
"/home/martin/PythonBeginner/Lesson9/Solved/TestDir"
"/home/martin/PythonBeginner/Lesson9/Solved/ToBe_Created.txt"
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/report_2017.xlsx ...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/report_2017.xls ...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/summary_2017.pptx ...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/outlook_2018.pptx ...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/template_overtime_increa
...
Creating file
```

```
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/icon_12123.jpg ...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/smile.gif ...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/paths.png ...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/workflow_improvement.png
...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/testing_process.pdf
...
Creating file
/home/martin/PythonBeginner/Lesson9/Solved/TestDir/report_2016.pdf ...
DONE
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

File Clean Up 1

Create script that will distribute all files of the below formats in a given folder as follows:

1. All **png, jpg, jpeg, tiff, gif** files into the folder **Images**
2. All **doc, docx, odt, txt, pdf** files into the folder **Docs**
3. All **xls,xlsx files** into the folder **Tables**
4. All **ppt, pptx** files into the folder **Presentations**

The Images, Docs, Tables, Presentations folders should be created inside the current working directory.

The script should be launched from the terminal (command line) providing it path to the directory, we want to tidy up.

Once the script has finished, it should print out a summary report, listing, how many items have been moved into each folder.

Example of use:

```
$ python cleanup1.py "/home/martin/PythonBeginner/Lesson9/Solved/TestDir"
MOVED
===============
Images         4
Tables         2
Docs           3
Presentations  2
```

Use the try except statement to handle situation, when the given folder will not be found:

```
$ python distribute.py "/home/martin/PythonBeginner/Lesson19/TestDir"
Folder "/home/martin/PythonBeginner/Lesson9/TestDir" does not exist
```

Also, when you will try to create new directories (Images, Tables,...) take care of possible error that will be thrown if that directory already exists.

To move the files you can use `shutil.move()` function.

You can test your script on directory `TestDir`, you have created in the previous task.

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

CSV

We will work with the file `example.csv`. You can download it here

General workflow:

1. Open a file
2. Create csv reader or csv writer object
3. Write or read the content

First, we will import the `csv` module:

```
>>> import csv
```

Opening a csv file

We first have to create a file object:

```
>>> file = open('example.csv', 'r+', newline='')
```

Reading a csv file

Now we can create a csv reader object using `csv.reader()` function:

```
>>> reader = csv.reader(file)
```

The csv reader object is an iterator, therefore we cannot see its content at once, we need to request it iteratively by using `next()` function or `for` loop:

```
>>> for row in reader:
...     print(row)
...
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', '']
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male']
['Murphy', 'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female']
['Cook', 'Floyd', 'Cook, Floyd', '28', '', 'Tester', 'Male']
['Glenn', 'Taylor', 'Glenn, Taylor', '35', 'Birmingham', 'Manager',
'Female']
['Mills', 'Amanda', 'Mills, Amanda', '41', 'Leicester', 'Quality
Assurance', 'Female']
['Harris', 'Roy', 'Harris, Roy', '22', 'London', 'Junior Programmer',
'Male']
['Chesterfield', 'Mark', 'Chesterfield, Mark', '46', 'Liverpool', 'SCRUM
Master', 'Male']
['Hammet', 'Sandra', 'Hammet, Sandra', '48', 'Liverpool', 'Designer',
'Female']
['Galagher', 'Fred', 'Galagher, Fred', '38', 'London', 'Programmer',
'Male']
['Murphy', 'John', 'Murphy, John', '35', 'London', 'Programmer', 'Male']
['Higgins', 'Mary', 'Higgins, Mary', '26', 'Leicester', 'Supervisor',
'Female']
```

Or we can convert it into a list. We however have to bare in mind, that we need to return at the beginning of the file, we have just read, otherwise we get an empty list:

```
>>> data = list(reader)
>>> data
[]
```

```
>>> file.seek(0)
0
>>> data = list(reader)
>>> data
[['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender'],
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', ''],
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male'], ['Murphy',
'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female'], ['Cook',
'Floyd', 'Cook, Floyd', '28', '', 'Tester', 'Male'], ['Glenn', 'Taylor',
'Glenn, Taylor', '35', 'Birmingham', 'Manager', 'Female'], ['Mills',
'Amanda', 'Mills, Amanda', '41', 'Leicester', 'Quality Assurance',
'Female'], ['Harris', 'Roy', 'Harris, Roy', '22', 'London', 'Junior
Programmer', 'Male'], ['Chesterfield', 'Mark', 'Chesterfield, Mark',
'46', 'Liverpool', 'SCRUM Master', 'Male'], ['Hammet', 'Sandra', 'Hammet,
Sandra', '48', 'Liverpool', 'Designer', 'Female'], ['Galagher', 'Fred',
'Galagher, Fred', '38', 'London', 'Programmer', 'Male'], ['Murphy',
'John', 'Murphy, John', '35', 'London', 'Programmer', 'Male'],
['Higgins', 'Mary', 'Higgins, Mary', '26', 'Leicester', 'Supervisor',
'Female']]
```

If a csv file is big, it is not convenient to convert the reader object into a list as it would take too much memory. It is preferred to use tools like `filter()` or `map()` to manipulate the data.

Code Task

1. Extract an arbitrary column from the reader object

2. Extract only those rows, that contain Londoners.

Writing into a csv file

We have the following employee record and we would like to add them into the `example.csv` file.

```
new_employee = ['Johnson', 'Greg', 'Johnson, Greg', '33', 'London',
'Senior Programmer', 'Male']
```

And now the company acquired new subsidiary, which has the following employees. We would like to add them into our csv file as well.

```
subsidiary = [['Carter', 'Philip', 'Carter, Philip', '20',
'Manchester', 'Senior Programmer', 'Male'],
              ['Hammond', 'Bob', 'Hammond, Bob', '43', 'Manchester',
'Senior Programmer', 'Male'],
              ['Taylor', 'Greg', 'Taylor, Greg', '54', 'Manchester',
'Senior Programmer', ''],
              ['Weaver', 'Mary', 'Weaver, Mary', '48', 'Manchester',
'Senior Programmer', 'Female']]
```

In order to write data into a csv file, we need to create a csv writer object first:

```
>>> writer = csv.writer(file)
```

We first make sure, the file cursor is located at the end of the file:

```
>>> file.seek(0,2)
```

To write a single row into our file, we use `writer.writerow()` function:

```
>>> writer.writerow(new_employee)
63
>>> file.flush()
```

We use the call `file.flush()` in order all the changes written into the file are reflected in it, before we close it.

To write multiple rows (list of lists) we use `writer.writerows()` function:

```
>>> writer.writerows(subsidiary)
>>> file.flush()
```

Reading data into a dictionary

Until now, the data we read or wrote into the file had form of list of lists. It is however possible to write or read data in form of a dictionary. First we will read the file `example.csv` into a dictionary structure:

```
>>> dict_reader = csv.DictReader(file)
>>> dict_reader.fieldnames
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
```

We can again retrieve the data using iteration protocol:

```
>>> next(dict_reader)
{'Surname': 'Smith', 'Name': 'John', 'Job': 'Programmer', 'Full Name':
'Smith, John', 'Age': '32', 'Gender': '', 'City': 'London'}
>>> next(dict_reader)
{'Surname': 'Doe', 'Name': 'Joe', 'Job': '', 'Full Name': 'Doe, Joe',
'Age': '34', 'Gender': 'Male', 'City': 'Liverpool'}
```

Writing dictionary into csv file

First create a writer object. To do that, we need a list of fieldnames first. We can retrieve them from a DictReader object:

```
>>> dict_reader.fieldnames
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
>>> writer = csv.DictWriter(file, dict_reader.fieldnames)
>>> file.seek(0,2)
1032
```

To write one row, we pass a dictionary into `writerow()` method:

```
1 new_employee = {'Surname': 'Moe', 'Name': 'Zoe', 'Job': '', 'Full
  Name': 'Moe, Zoe', 'Age': '34', 'Gender': 'Female', 'City':
  'Liverpool'}
```

To write multiple rows, we need a list of dictionaries for `writerows()` method:

```
new_employees = [{'Surname': 'Franz', 'Name': 'Ferdinand', 'Job': '',
'Full Name': 'Franz, Ferdinand', 'Age': '74', 'Gender': 'Male',
'City': 'Liverpool'},
{'Surname': 'Wilson', 'Name': 'Abel', 'Job': '', 'Full Name':
'Wilson, Abel', 'Age': '45', 'Gender': 'Male', 'City': 'Liverpool'}]
```

If you are asking, why we need a dictionary writer object, the answer is, that we will often have data in a dictionary-like format called JSON.

Requests

PYTHON ACADEMY / 11. FILE FORMATS / ONSITE EXERCISES / REQUESTS

In this section, we will learn, how to send and receive responses to and from a server. We will do it, using module called `requests`. This module is not a part of the standard library and therefore we should install it (if you do not have it installed already):

```
$ pip install requests
```

Now we can **import the module**:

```
>>> import requests
```

To ask server for a web page, we need to use so called **GET** method. This is represented by `requests.get()` function. We will request the content of the page called `example.com` and store the server response in variable `response`:

```
>>> response = requests.get('https://example.com')
```

Response object will consist of header and body. Body contains the actual content of `example.com`. The header contains the information about the page content.

Status Code

To check, what is the result of processing of our request on the server side we can look at the `status_code` attribute:

```
>>> response.status_code
200
```

Response

- Header - we can access header through the `header` attribute:

```
>>> response.headers
{'Cache-Control': 'max-age=604800', 'Content-Encoding': 'gzip'....
```

- Body - we access the content sent back the server by using the `response.text` attribute:

```
>>> response.text
'<!doctype html>\n<html>\n<head>\n    <title>Example Domain</title>\n\n
<meta charset="utf-8" />\n    <m
```

Original Request

The original request we have sento to the server is stored inside the `response.request` object.

- Header

```
>>> response.request.headers
{'Accept': '*/*', 'Connection': 'keep-alive', 'Accept-Encoding': 'gzip,
deflate', 'User-Agent': 'python-requests/2.18.2'}
```

- Body - there is no body in the GET message

```
>>> response.request.body
>>>
```

Post Method

We can send data to a server and create a record in its database using HTTP **POST** method

```
>>> url = https://requestb.in/14h8opb1
>>> resp = requests.post(url,data={"ts":time.time(),"message":"Hi
there"})
```

Response Body:

```
>>> resp.text
'ok'
```

Requests Body:

```
>>> resp.request.body
'ts=1507036488.91026&message=Hi+there'
>>>
```

JSON

JSON is a string that looks like a dictionary. It is often used for information transfer accross the internet.

We can usually obtain it from so called APIs.

```
>>> r = requests.get('http://api.openweathermap.org/data/2.5/weather?
q=Brno')
>>> print(r.text)
'{"cod":401, "message": "Invalid API key. Please see
http://openweathermap.org/faq#error401 for more info."}'
```

APIs often require you to be registered. Once you register yourself, they give you a API key, which you will always append to the request URL.

If you want to continue, you should register yourself **here**. Once you have your key, you can send the request as follows:

```
>>> r = requests.get('http://api.openweathermap.org/data/2.5/weather?
q=Brno&appid=HERE_GOES_YOUR_KEY')
```

The service on the other side have sent us JSON, we can access it via response object's `json()` function:

```
>>> r.json()
{'sys': {'id': 5899, 'country': 'CZ', 'sunset': 1507048010, 'message':
0.0022, 'type': 1, 'sunrise': 1507006623}, 'main': {'temp': 285.15,
'temp_max': 285.15, 'temp_min': 285.15, 'humidity': 93, 'pressure':
1014}, 'base': 'stations', 'dt': 1507035600, 'wind': {'deg': 160,
'speed': 3.6}, 'id': 3078610, 'clouds': {'all': 90}, 'coord': {'lon':
16.61, 'lat': 49.2}, 'name': 'Brno', 'cod': 200, 'visibility': 3800,
'weather': [{'id': 501, 'icon': '10d', 'description': 'moderate rain',
'main': 'Rain'}, {'id': 701, 'icon': '50d', 'description': 'mist',
'main': 'Mist'}]}
```

The `r.json()` call returns a dictionary, it means it has already parsed the original json string sent by the server.

This is the original json string:

```
>>> r.text
'{"sys": {"id": 5899, "country": "CZ", "sunset": 1507048010, "message":
0.0022, "type": 1, "sunrise": 1507006623}, "main": {"temp": 285.15,
"temp_max": 285.15, "temp_min": 285.15, "humidity": 93, "pressure":
1014}, "base": "stations", "dt": 1507035600, "wind": {"deg": 160,
"speed": 3.6}, "id": 3078610, "clouds": {"all": 90}, "coord": {"lon":
16.61, "lat": 49.2}, "name": "Brno", "cod": 200, "visibility": 3800,
"weather": [{"id": 501, "icon": "10d", "description": "moderate rain",
"main": "Rain"}, {"id": 701, "icon": "50d", "description": "mist",
"main": "Mist"}]}'
```

Module json

We can parse json string into a dictionary and back, using `json` module:

```
>>> import json
```

We can convert the json sent by the weather API using `json.loads()` function:

```
>> weather_dict = json.loads(r.text)
>>> weather_dict
{'sys': {'id': 5899, 'country': 'CZ', 'sunset': 1507048010, 'message':
0.0022, 'type': 1, 'sunrise': 1507006623}, 'main': {'temp': 285.15,
'temp_max': 285.15, 'temp_min': 285.15, 'humidity': 93, 'pressure':
1014}, 'base': 'stations', 'dt': 1507035600, 'wind': {'deg': 160,
'speed': 3.6}, 'id': 3078610, 'clouds': {'all': 90}, 'coord': {'lon':
16.61, 'lat': 49.2}, 'name': 'Brno', 'cod': 200, 'visibility': 3800,
'weather': [{'id': 501, 'icon': '10d', 'description': 'moderate rain',
'main': 'Rain'}, {'id': 701, 'icon': '50d', 'description': 'mist',
'main': 'Mist'}]}
```

To convert the dictionary back to json, we use `json.dumps()` function. We have used `indent` parameter inside the function call, to tell it, we would like to have each nested dictionary/json indented by 4 spaces. You will appreciate the result, once you print the resulting json string:

```
>>> weather_json = json.dumps(weather_dict, indent=4)
>>> print(weather_json)
```

Writing json into files

We can write or read json directly from a file using `json.dump()` and `json.load()` functions:

- Write

```
>>> file = open('test.json','w+')
>>> json.dump(weather_dict,file)
>>> file.flush()
>>> file.seek(0)
0
>>> file.read()
...Here you should see the string...
```

- Read

```
>>> file.seek(0)
0
>>> new_d= json.load(file)
>>> new_d
```

```
{'sys': {'id': 5899, 'message': 0.0022, 'sunset': 1507048010, 'country':
'CZ', 'type': 1, 'sunrise': 1507006623}, 'main': {'temp': 285.15,
'temp_max': 285.15, 'temp_min': 285.15, 'humidity': 93, 'pressure':
1014}, 'base': 'stations', 'dt': 1507035600, 'wind': {'deg': 160,
'speed': 3.6}, 'id': 3078610, 'clouds': {'all': 90}, 'coord': {'lon':
16.61, 'lat': 49.2}, 'name': 'Brno', 'cod': 200, 'visibility': 3800,
'weather': [{'id': 501, 'icon': '10d', 'description': 'moderate rain',
'main': 'Rain'}, {'id': 701, 'icon': '50d', 'description': 'mist',
'main': 'Mist'}]}
```

In General

File format defines the way, how data is encoded into a file. We can determine, what format a file is encoded in by checking its extension for example. Till now, in our tasks we have been using the `.txt` extension for plain text files (e.g. `test.txt`).

There are plenty of file formats out there. Some file formats store image information (.png, jpeg, etc.), others music (.mp3), some contain formatted textual information (.doc, docx).

In this lesson, we will be interested in file formats, that represent **text files of a given structure**. These file formats are

- `.csv`,
- `.json`

We will learn about these as they are very easy to work with as well as they are often used in the real life.

What is CSV?

CSV (comma-separated values) stores tabular data - that means data, organized into tables. You probably already know similar file format - xlsx (MS Excel). The difference is that CSV stores the data in a text form. CSV is therefore much more universal and you can open it in any text editor (such as Notepad), but also in a tabular processor (such as MS Excel). On the top of that, it's easy to work with in Python! And that's where we come in :)

How does CSV look

The data in columns are separated by a **delimiter**. Most common delimiter is a **comma** (,) or a semicolon (;). This delimiter has to be the same across the whole file. New rows can by simply created by pressing enter (adding `\n`). The file can look as follows:

```
Surname,Name,Full Name,Age,City
Smith,John,"Smith, John",32,London
Doe,Joe,"Doe, Joe",34,Liverpool
```

Task

Copy our example CSV file into your own **.csv** file that you'll create. Save it and close it. After you can open it in you tabular processor (MS Excel) and add some new rows and colums. Finally, open the CSV file in a text editor again to see the changes.

More info

On the first line of the file a header can be included with a list of the column names. This is optional, but strongly recommended. It allows the file to be self-documenting.

For more details regarding csv file format see CSV Wikipedia page

CSV Dialects

There is no well-defined standard for comma-separated value files. Therefore, different csv files might differ in certain things.

Some csv files might:

- use different delimiters
- may omit headers
- escape special characters in different manners

Combination of the above characteristics makes up so called **dialect**. People in different countries around the world, using tools as MS Excel or LibreOffice Calc, may have different setup of csv dialects.

As there are plenty options of how a csv file could look like, we may get unpredictable results depending on a person, that sends us the file if we would try to simply read that file and split it on comma delimiter.

Due to this complexity in csv structuring a **module called csv** has been implemented. Now we will learn why and how to work with it.

Reading CSV file

Module called **csv** is a part of standard Python library as modules **random** or **os** are. This module is here to help us to read as well as to write into csv files. As mentioned earlier, it is not an easy job, to recognize csv Dialect of a given file. This is where the **csv** module comes to help.

We load the module into our program with the following simple import:

```
import csv
```

To read a csv file in our Python programs, we first need to open the file. We use the classic built-in `open()` function. It is recommended to use `newline=''` parameter of the `open()`

function. This is due to difference in line termination symbols used in Windows vs. Unix-like (Linux, Mac) operating systems.

```
>>> file = open('example.csv', newline='')
```

We will use file called `example.csv`, that has the following content:

```
1 Surname,Name,Full Name,Age,City,Job,Gender
2 Smith,John,"Smith, John",32,London,Programmer,
3 Doe,Joe,"Doe, Joe",34,Liverpool,,Male
```

Once we have the file open, we need to create a special **csv reader** object from it using `csv.reader()` function. Do not forget to import the `csv` module first:

```
>>> import csv
>>> reader = csv.reader(file)
>>> reader
<_csv.reader object at 0x7f3fa459cf98>
```

The `csv.reader` object is an iterator. We will discuss more about iterator in the lesson about **Iteration Protocol**. For now, all you need to know is that if you pass interator object into the function `next()` we retrieve the next row of the table. However, we can of course also use it in any function or for loop.

Retrieving rows with `next()` will then look like:

```
>>> next(reader)
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
>>> next(reader)
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', '']
>>> next(reader)
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male']
```

As said we can retreive rows with for loop as well. But first we shouldn't forget we need to get to the beginning of the file, because we have already read all the content:

```
>>> for row in reader:
...     print(row)
...
```

```
>>> file.seek(0)
>>> for row in reader:
...     print(row)
...
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', '']
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male']
>>>
```

We can see that on each iteration, the reader object returns a list representing the table's row. Reading is about accessing the data in a target file and this is what we have achieved in the above described steps.

Often it is of course better to use **context manager** within which we perform operations on the csv file. The above steps serve as a demonstration, how data is read from a csv file:

```python
import csv

with open('example.csv', newline='') as f:
    reader = csv.reader(f)
    for row in reader:
        print(row)
```

Writing into CSV file

PYTHON ACADEMY / 11. FILE FORMATS / CSV / WRITING INTO CSV FILE

To write into a csv file, we need to create a **csv writer object** instead of reader object. The writer object provides two writing methods:

1. writerow()

The first method is `writerow()` and it allows us to write a single row into the table. For example, we have collected data about a new employee and we would like to enter them into our csv file.

```
>>> new_employee=['Mills','Amanda','Mills, Amanda',
41,'Leicester','Quality Assurance','Female']
```

The best way, how to do that is to open our file in append `a` mode, in order to keep the original content:

```
>>> with open('example.csv','a',newline='') as file:
...     writer = csv.writer(file)
...     writer.writerow(new_employee)
...
68
```

The steps to write into a file are:

1. open a file
2. create a write object from it
3. write into the file through the writer object

Now we can read the contents of the file as follows:

```
>>> with open('example.csv',newline='') as file:
...     reader = csv.reader(file)
...     for row in reader:
...             print(row)
...
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', '']
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male']
['Murphy', 'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female']
['Cook', 'Floyd', 'Cook, Floyd', '28', '', 'Tester', 'Male']
['Glenn', 'Taylor', 'Glenn, Taylor', '35', 'Birmingham', 'Manager',
'Female']
['Mills', 'Amanda', 'Mills, Amanda', '41', 'Leicester', 'Quality
Assurance', 'Female']
```

2. writerows()

The other writer object method me mentioned was `writerows()`. This method allows us to pass inside a list of lists - in other words a list of table rows. The method takes care of writing all the rows into our csv file.

We have three new employees to be entered into the table:

```
>>> employees = [['Harris','Roy','Harris, Roy',22,'London','Junior
Programmer','Male'],
...             ['Chesterfield','Mark','Chesterfield,
Mark',46,'Liverpool','SCRUM Master','Male'],
...             ['Hammet','Sandra','Hammet,
Sandra',48,'Liverpool','Designer','Female']]
```

We will add them using `writerows()`:

```
>>> with open('example.csv','a',newline='') as file:
...     writer = csv.writer(file)
...     writer.writerows(employees)
...
```

The operation has been performed successfully:

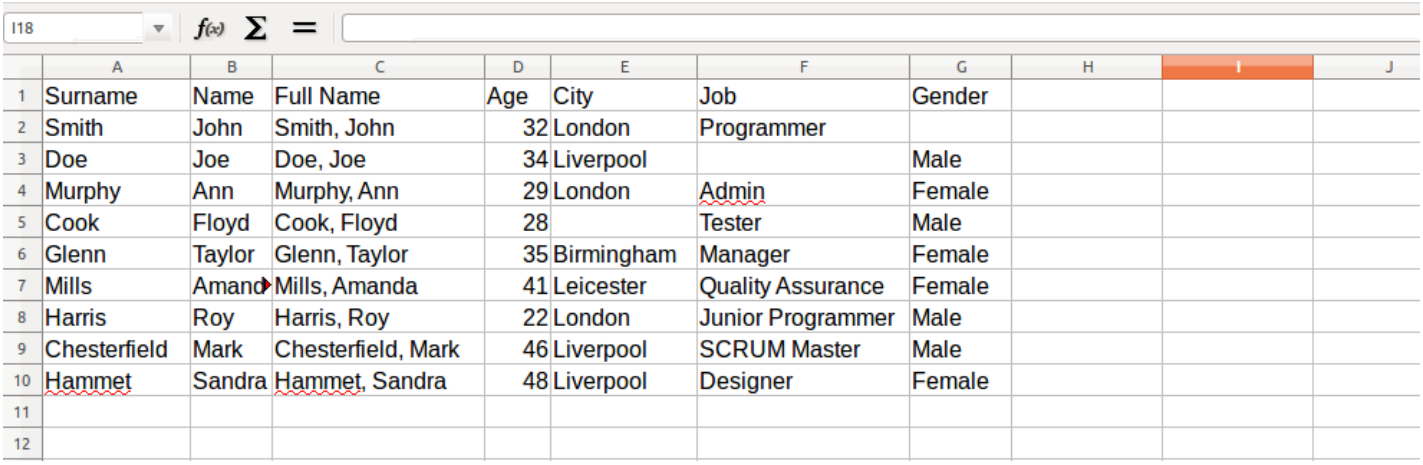
```
>>> with open('example.csv',newline='') as file:
...     reader=csv.reader(file)
...     for row in reader:
...             print(row)
...
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', '']
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male']
['Murphy', 'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female']
['Cook', 'Floyd', 'Cook, Floyd', '28', '', 'Tester', 'Male']
['Glenn', 'Taylor', 'Glenn, Taylor', '35', 'Birmingham', 'Manager',
'Female']
['Mills', 'Amanda', 'Mills, Amanda', '41', 'Leicester', 'Quality
Assurance', 'Female']
['Harris', 'Roy', 'Harris, Roy', '22', 'London', 'Junior Programmer',
'Male']
```

```
['Chesterfield', 'Mark', 'Chesterfield, Mark', '46', 'Liverpool', 'SCRUM
Master', 'Male']
['Hammet', 'Sandra', 'Hammet, Sandra', '48', 'Liverpool', 'Designer',
'Female']
```

Viewing our csv File

PYTHON ACADEMY / 11. FILE FORMATS / CSV / VIEWING OUR CSV FILE

After all the changes we have performed to our original `example.csv` file, we would like to be able to see the results in the MS Excel or LibreOffice Calc. And it is true, that all the changes have been correctly written and so they can be displayed with no inconveniences in the above mentioned programs:



	A	B	C	D	E	F	G	H	I	J
1	Surname	Name	Full Name	Age	City	Job	Gender			
2	Smith	John	Smith, John	32	London	Programmer				
3	Doe	Joe	Doe, Joe	34	Liverpool		Male			
4	Murphy	Ann	Murphy, Ann	29	London	Admin	Female			
5	Cook	Floyd	Cook, Floyd	28		Tester	Male			
6	Glenn	Taylor	Glenn, Taylor	35	Birmingham	Manager	Female			
7	Mills	Amand	Mills, Amanda	41	Leicester	Quality Assurance	Female			
8	Harris	Roy	Harris, Roy	22	London	Junior Programmer	Male			
9	Chesterfield	Mark	Chesterfield, Mark	46	Liverpool	SCRUM Master	Male			
10	Hammet	Sandra	Hammet, Sandra	48	Liverpool	Designer	Female			
11										
12										

Reading with Dict

PYTHON ACADEMY / 11. FILE FORMATS / CSV / READING WITH DICT

Until now, all the examples have show the read and write operations using lists. The **csv** module offers the possibility to perform these operations also using dictionaries.

To read the data out of a csv file into a dictionary, we need to create a `csv.DictReader` object:

```
>>> file = open('example.csv',newline='')
>>> reader = csv.DictReader(file)
```

Since Python 3.6, **reader** object returns **OrderedDict** object, which we can work with the same way as the normal dictionary.

```
>>> import csv
>>> file = open('example.csv',newline='')
>>> reader = csv.DictReader(file)
>>> for row in reader:
...     print(row)
...
OrderedDict([('Surname', 'Smith'), ('Name', 'John'), ('Full Name',
'Smith, John'), ('Age', '32'), ('City', 'London'), ('Job', 'Programmer'),
('Gender', '')])
OrderedDict([('Surname', 'Doe'), ('Name', 'Joe'), ('Full Name', 'Doe,
Joe'), ('Age', '34'), ('City', 'Liverpool'), ('Job', ''), ('Gender',
'Male')])
OrderedDict([('Surname', 'Murphy'), ('Name', 'Ann'), ('Full Name',
'Murphy, Ann'), ('Age', '29'), ('City', 'London'), ('Job', 'Admin'),
('Gender', 'Female')])
```

As we can see, there is no more header row in the ouput. Header names are inlcuded as the keys in each OrderedDict object. Each OrderedDict represents one row.

Writing with Dict

PYTHON ACADEMY / 11. FILE FORMATS / CSV / WRITING WITH DICT

In order to write a dictionary into a csv file, we need to create a `csv.DictWriter` object. To create it, we need to provide

- the file object
- column names (that will be used as dictionary keys)

We have a new employee, whose record we keep in a dictionary

```
>>> new_employee
{'Job': 'Programmer', 'Age': 38, 'Full Name': 'Galagher, Fred', 'City':
'London', 'Surname': 'Galagher', 'Gender': 'Male', 'Name': 'Fred'}
```

First, we need to get the column names. Therefore we open the file, extract column names with the method `.readline()` and close the file:

```
>>> file = open
>>> header = file.readline()
>>> header
'Surname,Name,Full Name,Age,City,Job,Gender\n'
>>> file.close()
```

We see that we need to strip the newline character `'\n'` as well as split the string into a list:

```
>>> header = header.strip('\n').split(',')
>>> header
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
```

We would like to write it into our `example.csv` file. Which we will open in `a+` mode. We do this in order to be able to extract the names of columns which we will need later.

```
>>> file = open('example.csv','a+',newline='')
```

And now we can create the `csv.DictWriter` object:

```
>>> writer = csv.DictWriter(file,header)
```

Since Python 3.6, in order to include header fields in our csv file, we need to first use the following method call:

```
writer.writeheader()
```

So if you are writing data into an empty file do not forget to call the `writeheader()` method.

Then we can begin to write the actual data into the file:

```
>>> writer.writerow(new_employee)
58
```

And now we can check, whether the data has been correctly entered:

```
>>> reader = csv.reader(file)
>>> for row in reader:
...     print(row)
...
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', '']
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male']
['Murphy', 'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female']
['Cook', 'Floyd', 'Cook, Floyd', '28', '', 'Tester', 'Male']
['Glenn', 'Taylor', 'Glenn, Taylor', '35', 'Birmingham', 'Manager',
'Female']
['Mills', 'Amanda', 'Mills, Amanda', '41', 'Leicester', 'Quality
Assurance', 'Female']
['Harris', 'Roy', 'Harris, Roy', '22', 'London', 'Junior Programmer',
'Male']
['Chesterfield', 'Mark', 'Chesterfield, Mark', '46', 'Liverpool', 'SCRUM
Master', 'Male']
['Hammet', 'Sandra', 'Hammet, Sandra', '48', 'Liverpool', 'Designer',
'Female']
['Galagher', 'Fred', 'Galagher, Fred', '38', 'London', 'Programmer',
'Male']
```

Method `DictWriter.writerows()` will require a sequence of dictionaries:

```
>>> new_employees
[{'Job': 'Programmer', 'Age': 35, 'Full Name': 'Murphy, John', 'City':
'London', 'Surname': 'Murphy', 'Gender': 'Male', 'Name': 'John'},
{'Job': 'Supervisor', 'Age': 26, 'Full Name': 'Higgins, Mary', 'City':
'Leicester', 'Surname': 'Higgins', 'Gender': 'Female', 'Name': 'Mary'}]
>>> writer.writerows(new_employees)
```

```
>>> file.seek(0)
0
>>> for row in reader:
...     print(row)
...
```

```
['Surname', 'Name', 'Full Name', 'Age', 'City', 'Job', 'Gender']
['Smith', 'John', 'Smith, John', '32', 'London', 'Programmer', '']
['Doe', 'Joe', 'Doe, Joe', '34', 'Liverpool', '', 'Male']
['Murphy', 'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female']
['Cook', 'Floyd', 'Cook, Floyd', '28', '', 'Tester', 'Male']
['Glenn', 'Taylor', 'Glenn, Taylor', '35', 'Birmingham', 'Manager',
'Female']
['Mills', 'Amanda', 'Mills, Amanda', '41', 'Leicester', 'Quality
Assurance', 'Female']
['Harris', 'Roy', 'Harris, Roy', '22', 'London', 'Junior Programmer',
'Male']
['Chesterfield', 'Mark', 'Chesterfield, Mark', '46', 'Liverpool', 'SCRUM
Master', 'Male']
['Hammet', 'Sandra', 'Hammet, Sandra', '48', 'Liverpool', 'Designer',
'Female']
['Galagher', 'Fred', 'Galagher, Fred', '38', 'London', 'Programmer',
'Male']
['Murphy', 'John', 'Murphy, John', '35', 'London', 'Programmer', 'Male']
['Higgins', 'Mary', 'Higgins, Mary', '26', 'Leicester', 'Supervisor',
'Female']
```

It will be very useful to read and write dictionaries out and into csv files once we will discover JSON file format.

Extracting a Column

When working with tabular data, we often need to extract individual columns. Below we have our table stored in the file `example.csv` .

```
Surname,Name,Full Name,Age,City,Job,Gender
Smith,John,"Smith, John",32,London,Programmer,
Doe,Joe,"Doe, Joe",34,Liverpool,,Male
```

To filter the column we can use the for loop and indexing:

```
>>> import csv
>>> ages_filtered = []
>>> file = open('example.csv', newline='')
>>> reader = csv.reader(file)
>>> for row in reader:
...     ages_filtered.append(row[3])
>>> print(ages_filtered)
['Age', '32', '34']
```

In case we would like to extract another column from the same file, we need to return to the beginning of the file first and then we can extract the `City` column for example:

```
>>> cities_filtered = []
>>> file.seek(0)
0
>>> for row in reader:
```

```
...     cities_filtered.append(row[4])
>>> print(cities_filtered)
['City', 'London', 'Liverpool']
```

Without going to the beginning to the file, we would get an empty list:

```
>>> for row in reader:
...     cities_filtered.append(row[4])
>>> print(cities_filtered)
[]
```

Sneak peak of functional tools

We could also use so called functional programming toolsuch as map() in combination with anonymous functions. You don't need to pay to much attetintion to these now, as you will learn these in the future lessons. But just to give you a little taste how much easier it can get ;)

Let's say we wanted to extract the entire column `Age` . Column `Age` is the fourth column (index 3 in a list) and therefore to extract it, we can use indexing operation on each item (row) of the table. We will represent this operation using **lambda** function `lambda x: x[3]` :

```
>>> file = open('example.csv', newline='')
>>> reader = csv.reader(file)
>>> ages_filtered = map(lambda x: x[3],reader)
>>> list(ages_filtered)
['Age', '32', '34']
```

The code `map(lambda x: x[3],reader)` basically says: **"For every element in reader, take the item located at the index 3."**

Filtering Rows

PYTHON ACADEMY / 11. FILE FORMATS / CSV + / FILTERING ROWS

Another task, that we may want to perform quite often is extraction of rows, based on a given condition.

We will work the file `example.csv`, where we have added three more rows:

```
Surname,Name,Full Name,Age,City,Job,Gender
Smith,John,"Smith, John",32,London,Programmer,
Doe,Joe,"Doe, Joe",34,Liverpool,,Male
Murphy,Ann,"Murphy, Ann",29,London,Admin,Female
Cook,Floyd,"Cook, Floyd",28,,Tester,Male
Glenn,Taylor,"Glenn, Taylor",35,Birmingham,Manager,Female
```

We want to extract only rows, representing females:

```
>>> gender_filtered = []
>>> file = open('example.csv', newline='')
>>> reader = csv.reader(file)
>>> for row in reader:
...     if row[-1] == 'Female':
...         gender_filtered.append(row)
>>> print(gender_filtered)
[['Murphy', 'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female'],
['Glenn', 'Taylor', 'Glenn, Taylor', '35', 'Birmingham', 'Manager',
'Female']]
>>> file.close()
```

Sneak peak of functional tools

Again the operation could be simpler with functional tool filter().

```
>>> gender_filtered = []
>>> file = open('example.csv', newline='')
>>> reader = csv.reader(file)
>>> females = list(filter(lambda x: x[-1]=='Female',reader))
>>> females
[['Murphy', 'Ann', 'Murphy, Ann', '29', 'London', 'Admin', 'Female'],
['Glenn', 'Taylor', 'Glenn, Taylor', '35', 'Birmingham', 'Manager',
'Female']]
>>> file.close()
```

csv Dialect Continued

As already mentioned, there is no well-defined standard for comma-separated value files. There are many parameters to control how csv parses or writes data. Rather than passing each of these parameters to the reader and writer separately, they can be grouped together into a **dialect object**. So the dialect tells our csv reader and writer objects, how a given csv file should be parsed.

To get the list of currently available dialects, use `csv.list_dialects()` function:

```
>>> print(csv.list_dialects())
['unix', 'excel-tab', 'excel']
```

The excel dialect is for working with data in the default export format for Microsoft Excel, and also works with LibreOffice

The unix dialect quotes all fields with double-quotes.

Creating our own Dialect

It is possible that we will have to work with files where the data is separated not by comma, but caret. In that case we have to create a new dialect- rules, how this specific file should be read.

The dialect is created using the `csv.register_dialect()` method

Inside the `register_dialect()` function, the folowing parameters can be specified:

Parameter	Default Value	Meaning
delimiter	,	Field separator (one character)

Parameter	Default Value	Meaning
doublequote	`True`	Flag controlling whether quotechar instances are doubled
escapechar	`None`	Character used to indicate an escape sequence
lineterminator	`\r\n`	String used by writer to terminate a line
quotechar	`"`	String to surround fields containing special values (one character)
quoting	`QUOTE_MINIMAL`	discussed in the above section
skipinitialspace	`False`	Ignore whitespace after the field delimiter

And this is how we register the dialect - the first argument is the name of the dialect:

```
>>> csv.register_dialect('tabs',
                         delimiter = '\t',
                         lineterminator = '\n',
                         quotechar = "'",
                         quoting = csv.QUOTE_NONNUMERIC)
>>> csv.list_dialects()
['unix', 'excel-tab', 'excel', 'tabs']
```

If we decide to write our data into csv using this dialect, the columns will be delimited by tab, lines terminated by newline character and for quoting will be used single quote character. Quoted will be only non-numeric contents of cells.

```
>>> csv.list_dialects()
['unix', 'excel-tab', 'excel', 'tabs']
>>> file = open('test.csv', 'w+')
>>> data = [["Name", "Title", "Age"],
```

```
...     ["John", "machinist", 46],
...     ["Bob", "welder", 59],
...     ["Heinz", "planner",32]]
>>> writer = csv.writer(file, dialect = 'tabs')
>>> writer.writerows(data)
>>> file.seek(0)
0
>>> file.read()
"'Name'\t'Title'\t'Age'\n'John'\t'machinist'\t46\n'Bob'\t'welder'\t59\n'Hei
>>> file.seek(0)
0
>>> print(file.read())
'Name'  'Title' 'Age'
'John'  'machinist' 46
'Bob'   'welder'    59
'Heinz' 'planner'   32
```

You may have noticed that quoting changed from double to single quotes as well - as we have defined it in our dialect.

Quoting

PYTHON ACADEMY / 11. FILE FORMATS / CSV + / QUOTING

The `register_dialect()` function has parameter `quoting`. In the previous section, we have passed it a special value, without explaining what other values, does this parameter accept. This section will fill that gap.

Why quoting is so important?

If, for example, comma is used in our file as a delimiter and we want to include it in some cells as part of a string, we need to tell the parser that this comma is here as part of the data set. To do that, we need to enclose such cell values in quotes. Quoting is used to, so to say, escape special characters - characters that have special meaning for the csv parser.

There are 4 posible modes in which the quoting can be set up:

- **QUOTE_ALL** - writer object will quote everything, regardless of type.
- **QUOTE_MINIMAL** - writer object will quote fields with special characters (anything that would confuse a parser configured with the same dialect and options). This is the default
- **QUOTE_NONNUMERIC** - writer object will quote all fields that are not integers or floats. When used with the reader, input fields that are not quoted are converted to floats.
- **QUOTE_NONE** - Do not quote anything on output. When used with the reader, quote characters are included in the field values (normally, they are treated as delimiters and stripped).

These mode names have to be prepended with the csv classifier, e.g. `csv.QUOTE_NONNUMERIC`, because they belong to the csv module.

What is JSON?

JSON is an abbreviation for JavaScript Object Notation. JSON belongs to the wide scale of data formats as CSV, XML, HTML. JSON is a string that looks like Python dictionary:

```
'{"a": true, "c": null, "1": 2, "b": false}'
```

It provides convenient way for storing data we would store in dictionaries in our Python programs.

JSON is often used to transport data over the internet and therefore it is important to learn to work with it. Data packed as JSON is often returned by so called web applications' APIs. For example weather services, information about locations from Google Geocoding API etc.

Example of JSON sent back by Google Geocoding API:

```
'{"results": [{"geometry": {"bounds": {"northeast": {"lat":
49.29448499999999,
                                                    "lng": 16.7278532},
                                       "southwest": {"lat": 49.1096552,
                                                     "lng": 16.4280678}},
                            "viewport": {"northeast": {"lat":
49.29448499999999,
                                                       "lng": 16.7278532},
                                        "southwest": {"lat": 49.1096552,
                                                      "lng":
16.4280678}},
                            "location": {"lat": 49.1950602, "lng":
16.6068371},
                            "location_type": "APPROXIMATE"},
              "address_components": [{"long_name": "Brno",
                                      "types": ["locality", "political"],
                                      "short_name": "Brno"},
                                     {"long_name": "Brno-City District",
                                      "types":
["administrative_area_level_2", "political"],
```

```
                                    "short_name": "Brno-City District"},
                                    {"long_name": "South Moravian
Region",
                                    "types":
["administrative_area_level_1", "political"],
                                    "short_name": "South Moravian
Region"},
                                    {"long_name": "Czechia",
                                    "types": ["country", "political"],
                                    "short_name": "CZ"}],
            "types": ["locality", "political"],
            "formatted_address": "Brno, Czechia",
            "place_id": "ChIJEVE_wDqUEkcRsLEUZg-vAAQ"}],
 "status": "OK"}'
```

JSON Format Rules

PYTHON ACADEMY / 11. FILE FORMATS / JSON / JSON FORMAT RULES

The format of JSON encoding is almost identical to Python syntax except for a few minor changes:

- non-string dictionary keys are converted to strings
- `True` is mapped to `'true'` , `False` is mapped to `'false'`
- `None` is mapped to `null`
- **single quotes** on strings are converted **to double quotes**

Working with JSON

PYTHON ACADEMY / 11. FILE FORMATS / JSON / WORKING WITH JSON

First, we need to import module called **json**:

```
import json
```

In general, we can say our work with JSON will consist of:

1. data encoding into JSON format string - `json.dumps()`
2. writing encoded JSON string to a file
3. decoding from JSON format string (into dict)- `json.loads()`
4. or vice versa

We will start with the encoding part.

Encoding into JSON

To encode data into JSON format, we will need to use `json.dumps()` function. The `json.dumps()` function converts a dictionary into JSON string.

Therefore we will first create a dictionary:

```
>>> employee = {'Job': 'Programmer', 'Age': 38, 'Full Name': 'Galagher,
Fred', 'City': 'London', 'Surname': 'Galagher', 'Gender': 'Male', 'Name':
'Fred'}
```

Now we will convert it into JSON:

```
>>> import json
>>> data = json.dumps(employee)
>>> data
'{"Name": "Fred", "Job": "Programmer", "Full Name": "Galagher, Fred",
"Surname": "Galagher", "City": "London", "Gender": "Male", "Age": 38}'
```

Additionally, if our dictionary would include the values `True`, `False` or `None` the output in JSON would be `true`, `false` or `null` as this is how these values are represented in Javascript.

```
>>> employee = {'Active': False, 'Job': None, 'Age': 38, 'Full Name':
'Galagher, Fred', 'City': 'London', 'Surname': 'Galagher', 'Gender':
'Male', 'Name': 'Fred'}
```

Now we will convert it again into JSON:

```
>>> import json
>>> data = json.dumps(employee)
>>> data
'{"Active": false, "Job": null, "Age": 38, "Full Name": "Galagher, Fred",
"City": "London", "Surname": "Galagher", "Gender": "Male", "Name":
"Fred"}'
```

Decoding from JSON

PYTHON ACADEMY / 11. FILE FORMATS / JSON / DECODING FROM JSON

To convert the JSON string back to a dictionary, we use `json.loads()` function:

We have our `data` variable with information about an employee encoded into JSON string:

```
>>> data
'{"Name": "Fred", "Job": "Programmer", "Full Name": "Galagher, Fred",
"Surname": "Galagher", "City": "London", "Gender": "Male", "Age": 38}'
>>> type(data)
<class 'str'>
```

Now we will convert the string back to a dictionary:

```
>>> emp = json.loads(data)
>>> emp
{'Name': 'Fred', 'Job': 'Programmer', 'Full Name': 'Galagher, Fred',
'Surname': 'Galagher', 'City': 'London', 'Gender': 'Male', 'Age': 38}
>>> type(emp)
<class 'dict'>
```

Special Encoding Parameters

Till now, all our JSON strings were hard to read. The `json.dumps()` function offers few parameters to make JSON strings look nicer.

1. **Force indentation**

To distinguish among different nested structures in our JSON string, we can use the `indent` parameter:

- We have our employee dictionary:

```
>>> emp
{'Name': 'Fred', 'Job': 'Programmer', 'Full Name': 'Galagher, Fred',
'Surname': 'Galagher', 'City': 'London', 'Gender': 'Male', 'Age': 38}
>>> type(emp)
<class 'dict'>
```

- We convert it into JSON, using `indent=4` argument. This tells `json` to indent the nested structures by 4 spaces from the previous level:

```
>>> data = json.dumps(emp,indent=4)
>>> data
'{\n    "Name": "Fred",\n    "Job": "Programmer",\n    "Full Name":
"Galagher, Fred",\n    "Surname": "Galagher",\n    "City": "London",\n
"Gender": "Male",\n    "Age": 38\n}'
```

- Once we print the resulting string, it looks much nicer

```
>>> print(data)
{
    "Name": "Fred",
    "Job": "Programmer",
    "Full Name": "Galagher, Fred",
    "Surname": "Galagher",
    "City": "London",
    "Gender": "Male",
    "Age": 38
```

```
}
```

2. **Sort Keys**

Often it is more comfortable to read JSON, if the keys are sorted using `sort_keys=True` parameter:

```
>>> data = json.dumps(emp,indent=4,sort_keys=True)
>>> print(data)
{
    "Age": 38,
    "City": "London",
    "Full Name": "Galagher, Fred",
    "Gender": "Male",
    "Job": "Programmer",
    "Name": "Fred",
    "Surname": "Galagher"
}
```

Writing JSON into a file

PYTHON ACADEMY / 11. FILE FORMATS / JSON / WRITING JSON INTO A FILE

In case we would like our JSON to be directly written into a file, while encoding, we can use `json.dump()` function. This function is missing the `s` at the end of its name compared to `json.dumps()`. The latter, just converts a dictionary to a string meanwhile `json.dump()` also writes the encoded JSON string into a given file.

The file we want to write into, has to support writing operation (modes `w`, `a`, `w+`, `a+`, `r+`):

```
>>> file = open('employees.json','w')
>>> json.dump(employee, file, indent=4)
>>> file.close()
```

And the content of the file `employees.json` looks as follows:

```
{
```

```
    "Name": "Fred",
    "Job": "Programmer",
    "Full Name": "Galagher, Fred",
    "Surname": "Galagher",
    "City": "London",
    "Gender": "Male",
    "Age": 38
}
```

Reading JSON from a file

Similarly to dichotomy among `json.dumps()` and `json.dump()`, we have counterpart to `json.loads()` function. It is called `json.load()` and expects a file object as argument. This file object has to support read operation:

```
>>> file = open('employees.json')
>>> content=json.load(file)
>>> content
{'Name': 'Fred', 'Job': 'Programmer', 'Full Name': 'Galagher, Fred',
'Surname': 'Galagher', 'City': 'London', 'Gender': 'Male', 'Age': 38}
>>> type(content)
<class 'dict'>
```

The `json.load()` function not only reads the JSON string, but it also converts this string into a dictionary.

CSV

1/6

What is a csv file?

A. Text file, that represents tabular data by delimiting them into rows (file lines separated by `\n`) and columns (strings separated by a delimiter character)

B. Text file that represents data in a dictionary-like fashion

C. Text file that represents data in a hierarchical structure of tags

D. Text file that represents data by separating them into tags

JSON

1/7

What is a JSON file?

A. Text file that represents data using tags

B. File that stores JavaScript code

C. Text file that stores data points separated by commas

D. Text file that represents data in a dictionary-like fashion

Salary Database

To complete this task, you will need to **download** CSV file from this **link**. This file stores names of employees and their salaries. Your task is to get an employee with **highest salary** and employee with **lowest salary**. In addition to that, please come up with an average salary. Store this information in new CSV file. This file shall have 2 colums - first should be the information type (ex.: Minimum salary), second the number. This file shall have header. It might look like this:

TYPE	SALARY
Minimum salary	xxxx
Maximum salary	yyyyy
Average salary	zzzzz

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

Display Weather Data

This task includes working with following **JSON** file:

```
{
  "name":"Prague",
  "weather":[{"main":"Rain","description":"light intensity rain"}],
  "main":
{"temp":28,"pressure":1012,"humidity":81,"temp_min":27,"temp_max":28},
  "visibility":10000,
  "wind":{"speed":4.1,"deg":80},
  "clouds":{"all":90}
}
```

Please copy and save this JSON. Your task is to **gather some data from this file** and print them in a nice, formatted output. The output should look like this:

```
##############################
Prague
-------------
Rain
Current temerature:  28 C
Minimum temperature: 27 C
Maximum temperature: 28 C
Wind speed:          4.1 m/s
##############################
```

Code Solution

Use dropdown feature below if you want to see, how we wrote the code.

Click to see our solution

DALŠÍ LEKCE