File Handling (Introduction)

File handling in Python is a powerful and versatile tool that can be used to perform a wide range of operations.

The key function for working with files in Python is the `open()` function.

The `open()` function takes two parameters; *filename*, and *mode*.

There are four different methods (modes) for opening a file:

`"r"` - Read - Default value. Opens a file for reading, error if the file does not exist

`"a"` - Append - Opens a file for appending, creates the file if it does not exist

`"w"` - Write - Opens a file for writing, creates the file if it does not exist

`"x"` - Create - Creates the specified file, returns an error if the file exists

In addition, you can specify if the file should be handled as binary or text mode (**Type of Files**)

"t" - Text - Default value. Text mode

"b" - Binary - Binary mode (e.g. images)

Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

The code above is the same as:

```
f = open("demofile.txt", "rt")
```

Because "r" for read, and "t" for text are the default values, you do not need to specify them.

Open a file on a different location:

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

Return the 5 first characters of the file:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

Read one line of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
```

Read two lines of the file:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

Loop through the file line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

Close Files

It is a good practice to always close the file when you are done with it.

Close the file when you are finish with it:

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

Write to an Existing File
To write to an existing file, you must add a parameter to the open() function:

"a" - Append - will append to the end of the file

"w" - Write - will overwrite any existing content

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

Open the file "demofile3.txt" and overwrite the content:

```
f = open("demofile3.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the overwriting:
f = open("demofile3.txt", "r")
print(f.read())
```

Create a New File

To create a new file in Python, use the open() method, with one of the following parameters:

"x" - Create - will create a file, returns an error if the file exist

Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

Delete a File

To delete a file, you must import the OS module, and run its os.remove() function:

Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

Check if file exists, *then* delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```

Delete Folder

To delete an entire folder, use the os.rmdir() method:

Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Serialization in Python

Serialization refers to the process of converting an object into a format that can be easily stored, transmitted, or reconstructed later. In Python, this involves converting complex data structures, such as objects or dictionaries, into a byte stream.

Why Do We Use Serialization?

Serialization allows data to be easily saved to disk or transmitted over a network, and later reconstructed back into its original form. It is important for tasks like saving game states, storing user preferences, or exchanging data between different systems.

Serialization Libraries in Python

Python offers several libraries for serialization, each with its own advantages. Here is a detailed overview of some commonly used serialization libraries in Python −

- **Pickle −** This is Python's built-in module for serializing and deserializing Python objects. It is simple to use but specific to Python and may have security implications if used with untrusted data.
- **JSON −** JSON (JavaScript Object Notation) is a lightweight data interchange format that is human-readable and easy to parse. It is ideal for web APIs and cross-platform communication.
- **YAML −** YAML: YAML (YAML Ain't Markup Language) is a human-readable data serialization standard that is also easy for both humans and machines to read and write. It supports complex data structures and is often used in configuration files.

Serialization Using Pickle Module

The pickle module in Python is used for serializing and deserializing objects. Serialization, also known as **pickling**, involves converting a Python object into a byte stream, which can then be stored in a file or transmitted over a network.

Deserialization, or **unpickling**, is the reverse process, converting the byte stream back into a Python object.

Serializing an Object

We can serialize an object using the **dump()** function and write it to a file. The file must be opened in binary write mode ('wb').

```
import pickle

data = {'name': 'Alice', 'age': 30, 'city': 'New York'}
# Open a file in binary write mode
with open('data.pkl', 'wb') as file:

# Serialize the data and write it to the file

pickle.dump(data, file)

print ("File created!!")
```

When above code is executed, the dictionary object's byte representation will be stored in data.pkl file.

Deserializing an Object

To deserialize or unpickle the object, you can use the load() function. The file must be opened in binary read mode ('rb') as shown below-

```
import pickle

# Open the file in binary read mode

with open('data.pkl', 'rb') as file:

   # Deserialize the data

   data = pickle.load(file)

print(data)
```

This will read the byte stream from "data.pkl" and convert it back into the original dictionary as shown below –

```
{'name': 'Alice', 'age': 30, 'city': 'New York'}
```

JSON in Python

Python has a built-in package called json, which can be used to work with JSON data. Import the json module:

```
import json
```

Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the json.dumps() method.

```
import json

# a Python object (dict):
x = {
  "name": "John",
  "age": 30,
  "city": "New York"
}

# convert into JSON:
y = json.dumps(x)

# the result is a JSON string:
print(y)
```

Parse JSON - Convert from JSON to Python

If you have a JSON string, you can parse it by using the json.loads() method. The result will be a Python dictionary.

```
import json

# some JSON:
x =  '{ "name":"John", "age":30, "city":"New York"}'

# parse x:
y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

Convert Python objects into JSON strings, and print the values:

```
import json

print(json.dumps({"name": "John", "age": 30}))
print(json.dumps(["apple", "bananas"]))
print(json.dumps(("apple", "bananas")))
print(json.dumps("hello"))
print(json.dumps(42))
print(json.dumps(31.76))
print(json.dumps(True))
print(json.dumps(False))
print(json.dumps(None))
```

When you convert from Python to JSON, Python objects are converted into the JSON (JavaScript) equivalent:

Python	JSON
dict	Object
list	Array
tuple	Array
str	String
int	Number
float	Number
True	true
False	false
None	null

Convert a Python object containing all the legal data types:

```python
import json

x = {
  "name": "John",
  "age": 30,
  "married": True,
  "divorced": False,
  "children": ("Ann","Billy"),
  "pets": None,
  "cars": [
    {"model": "BMW 230", "mpg": 27.5},
    {"model": "Ford Edge", "mpg": 24.1}
  ]
}

print(json.dumps(x))
```

Output:

```
{"name": "John", "age": 30, "married": true, "divorced": false,
"children": ["Ann", "Billy"], "pets": null, "cars": [{"model":
"BMW 230", "mpg": 27.5}, {"model": "Ford Edge", "mpg":
24.1}]}
```

Format the Result

The example above prints a JSON string, but it is not very easy to read, with no indentations and line breaks.

The json.dumps() method has parameters to make it easier to read the result:

Use the indent parameter to define the numbers of indents:

```
json.dumps(x, indent=4)
```

Output:

```
{
    "name": "John",
    "age": 30,
    "married": true,
    "divorced": false,
    "children": [
        "Ann",
        "Billy"
    ],
    "pets": null,
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ]
}
```

You can also define the separators, default value is (", ", ": "), which means using a comma and a space to separate each object, and a colon and a space to separate keys from values:

Use the `separators` parameter to change the default separator:

```
json.dumps(x, indent=4, separators=(". ", " = "))
```

Output:

```
{
    "name" = "John".
    "age" = 30.
    "married" = true.
    "divorced" = false.
    "children" = [
        "Ann".
        "Billy"
    ].
    "pets" = null.
    "cars" = [
        {
            "model" = "BMW 230".
            "mpg" = 27.5
        }.
        {
            "model" = "Ford Edge".
            "mpg" = 24.1
        }
    ]
}
```

Order the Result

The json.dumps() method has parameters to order the keys in the result:

Use the sort_keys parameter to specify if the result should be sorted or not:

```
json.dumps(x, indent=4, sort_keys=True)
```

Output:

```
{
    "age": 30,
    "cars": [
        {
            "model": "BMW 230",
            "mpg": 27.5
        },
        {
            "model": "Ford Edge",
            "mpg": 24.1
        }
    ],
    "children": [
        "Ann",
        "Billy"
    ],
    "divorced": false,
    "married": true,
    "name": "John",
    "pets": null
}
```

Python CSV

The CSV (Comma Separated Values) format is a common and straightforward way to store tabular data. To represent a CSV file, it should have the .csv file extension.

```
SN, Name,    City
1,  Michael, New Jersey
2,  Jack,    California
```

We first need to import the module using:

```python
import csv
```

Read CSV Files with Python

The csv module provides the csv.reader() function to read a CSV file. Suppose we have a csv file named people.csv with the following entries.

```
Name,   Age, Profession
Jack,   23,  Doctor
Miller, 22,  Engineer
```

Now, let's read this csv file.

```python
import csv

with open('people.csv', 'r') as file:
    reader = csv.reader(file)
    for row in reader:
        print(row)
```

Output:

```
['Name', 'Age', 'Profession']
['Jack', '23', 'Doctor']
['Miller', '22', 'Engineer']
```

Write to CSV Files with Python

The csv module provides the csv.writer() function to write to a CSV file.

```python
import csv

with open('protagonist.csv', 'w', newline=' ') as file:

    writer = csv.writer(file)

    writer.writerow(["SN", "Movie", "Protagonist"])

    writer.writerow([1, "Lord of the Rings", "Frodo Baggins"])

    writer.writerow([2, "Harry Potter", "Harry Potter"])
```

File Content:

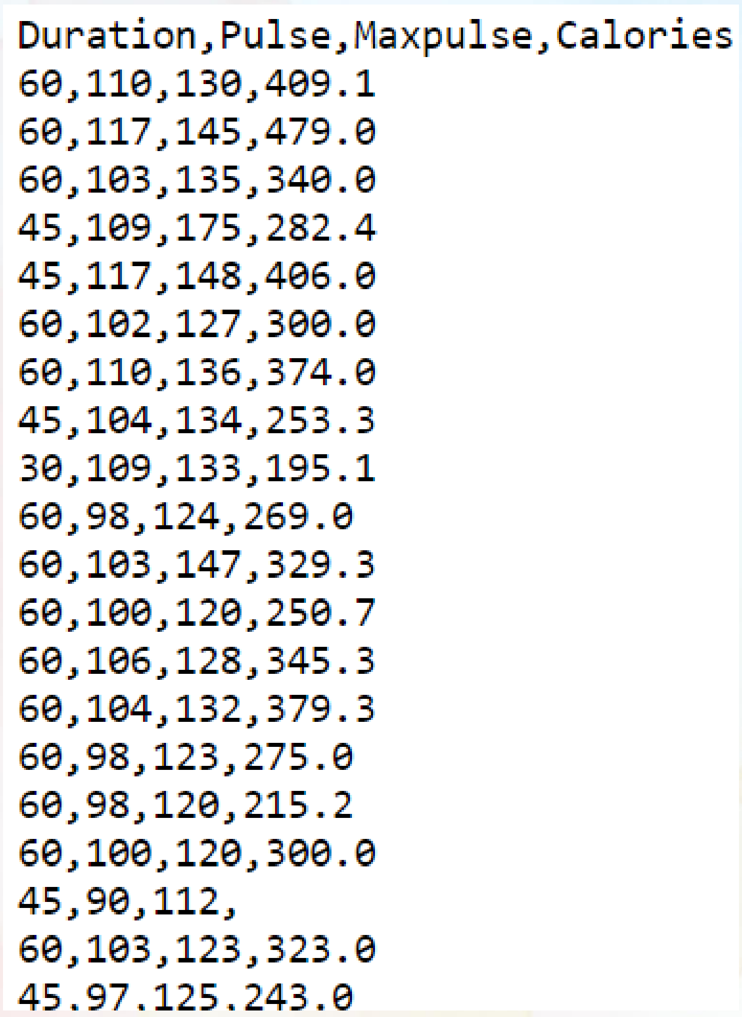
```
SN,Movie,Protagonist

1,Lord of the Rings,Frodo Baggins

2,Harry Potter,Harry Potter
```

CSV Files (Pandas)

A simple way to store big data sets is to use CSV files (comma separated files).

CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

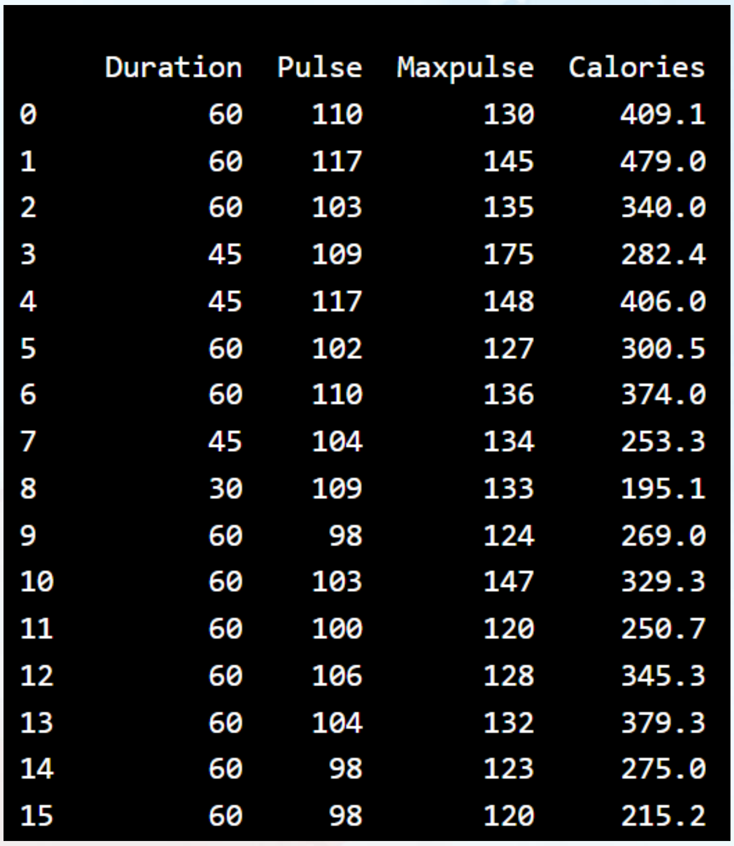
data.csv



```
Duration,Pulse,Maxpulse,Calories
60,110,130,409.1
60,117,145,479.0
60,103,135,340.0
45,109,175,282.4
45,117,148,406.0
60,102,127,300.0
60,110,136,374.0
45,104,134,253.3
30,109,133,195.1
60,98,124,269.0
60,103,147,329.3
60,100,120,250.7
60,106,128,345.3
60,104,132,379.3
60,98,123,275.0
60,98,120,215.2
60,100,120,300.0
45,90,112,
60,103,123,323.0
45,97,125,243.0
```

Load the CSV into a DataFrame:

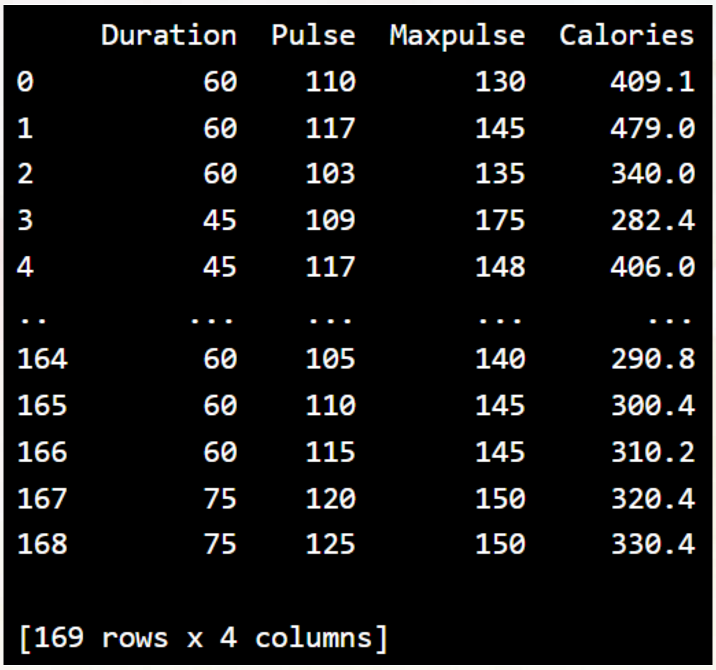
```
import pandas as pd

df = pd.read_csv('data.csv')

print(df.to_string())
```



```
    Duration  Pulse  Maxpulse  Calories
0         60    110       130     409.1
1         60    117       145     479.0
2         60    103       135     340.0
3         45    109       175     282.4
4         45    117       148     406.0
5         60    102       127     300.5
6         60    110       136     374.0
7         45    104       134     253.3
8         30    109       133     195.1
9         60     98       124     269.0
10        60    103       147     329.3
11        60    100       120     250.7
12        60    106       128     345.3
13        60    104       132     379.3
14        60     98       123     275.0
15        60     98       120     215.2
```

If you have a large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows:

```
import pandas as pd
df = pd.read_csv('data.csv')
print(df)
```



```
     Duration  Pulse  Maxpulse  Calories
0          60    110       130     409.1
1          60    117       145     479.0
2          60    103       135     340.0
3          45    109       175     282.4
4          45    117       148     406.0
..        ...    ...       ...       ...
164        60    105       140     290.8
165        60    110       145     300.4
166        60    115       145     310.2
167        75    120       150     320.4
168        75    125       150     330.4

[169 rows x 4 columns]
```

max_rows

The number of rows returned is defined in Pandas option settings.

You can check your system's maximum rows with the pd.options.display.max_rows statement.

Check the number of maximum returned rows:

```
import pandas as pd

print(pd.options.display.max_rows)
```

Output:

```
60
```

In my system the number is 60, which means that if the DataFrame contains more than 60 rows, the print(df) statement will return only the ***headers and the first and last 5 rows***.

Increase the maximum number of rows to display the entire DataFrame:

```
import pandas as pd

pd.options.display.max_rows = 9999

df = pd.read_csv('data.csv')

print(df)
```