JSON - Introduction

JSON → JavaScript Object Notation

- JSON → **J**ava**S**cript **O**bject ***N***otation

- JSON is a text-based and lightweight data format for storing and transporting data

- JSON is "self-describing" and easy to understand
- JSON is a subset of JavaScript
  - but is independent of any of the programming languages and almost all languages can easily analyse the text.
- JSON files have an extension of `.json`
- JSON files can be created using any programming language
- JSON format was originally specified by Douglas Crockford in early 2000

- Its unique properties like text-based, lightweight, language independence etc. make it an ideal candidate for the data-interchange operations

Use of JSON

- JSON is used to transfer the data from one system to another. It can transfer data between two computers, database, programs etc.

- It is used for transmitting serialized data over network connections.
- It can be used with all the major programming languages.
- Useful in data transfer from the web application to the server.
- Most of the web services use JSON based format for data transfer.

Properties of JSON

- It is a text-based lightweight data interchange format.
- It has been extended from the JavaScript language.
- JSON files use the extension *.json*.
- As the format is text-based it is easy to read and write by both the user/programmer and the machines.
- JSON is independent of programming language but uses the conventions that are well known within the C-Family of languages like C, C++, C#, JavaScript, Java, Python, Perl etc.

Syntax of JSON

- Basic syntax which is used in forming JSON.

- JSON built on two structural entities.
  - key-value pairs collections
    - object, strut, record, dictionary etc.

  - ordered list of values.
    - realized as array, list etc.

JSON Syntax Rules

- JSON object literals are surrounded by curly braces {}.
- JSON object literals contains ***key/value*** pairs.
- *Keys* and *values* are separated by a colon. ***key : value***
- *Keys* must be strings and must be enclosed in double quotes.
- Values must be a valid JSON data type:
  String, number, object, array, Boolean, null
- Each *key/value* pair is separated by a comma.
- Square brackets [] hold arrays

JSON Example - JSON car Object

- A car object with the following basic properties and attributes:

```
{
  "make": "Toyota",
  "year": 2019,
  "colour": "Green"
}
```

OR like this

```
{"make": "Toyota", "year": 2019, "colour": "Green"}
```

- It is a common mistake to call a JSON object literal "a JSON object".
- JSON cannot be an object.
- JSON is a string format.
- The data is only JSON when it is in a string format.

JSON Data Types

- In JSON, ***values*** must be one of the following data types:
  - a string
  - a number
  - an object (JSON object)
  - an array
  - a boolean
  - null

JSON Strings and JSON Numbers

- Strings in JSON must be written in double quotes.

```
{"name":"Aine"}
```

- Numbers in JSON must be an integer or a floating point.

```
{"age":21}
```

JSON Values

- Values in JSON can be objects.

```
{
 "employee":{"name":"Aine", "age":21}
}
```

- Values in JSON can be arrays.

```
{
     "employees":["Sean", "Aine", "Paul"]
}
```

JSON values

- Values in JSON can be true/false.

```
{"available":true}
```

- Values in JSON can be null.

```
{"middlename":null}
```

JSON student object

- Create a JSON object describing an “Student” with *first Name*, *last name* and *student id*:

```
{
    "fname" : "Snow",
    "lname" : "White",
    "idNo" : 123
}
```

Or can be written like this

```
{"fname":"Snow", "lname":"White", "idNo":123 }
```

JSON Arrays

- Arrays in JSON are an ordered collection of data.
- The array is enclosed in square brackets “[]”.
- The array elements are separated by a comma.

- A Student with three subjects.
- use an array to store subjects .

```
{
    "fname" : "Snow",
    "lname" : "White",
    "idNo" : 123,
    "subjects":["maths", "CAOS", "OOP"]
 }
```

JSON can have a nested structure

- JSON can have a nested structure,
  - a JSON object literal can have another JSON object contained inside it.
- This allows more complex data format.

- An array of employees and each employee can have a car.
- Create a nested Car JSON object inside the Employee JSON

oyees.json ↗ ...

```
[{
  "fName": "Oisin",
  "lName": "Ryan",
  "empId": 3456,
  "languages":["C#", "javaScript", "Python" ],
  "car":{
    "make":"Ford Mondeo",
    "year":2021,
    "colour": "Blue"
  }
},
{
  "fName": "Laoise",
  "lName": "McLaughlin",
  "empId": 123456,
  "languages":["java", "javaScript" ],
  "car":{
    "make":"Toyota Avensis",
    "year":2017,
    "colour": "Silver"
  }
}]
```

JSON Object Literals vs JSON string

- Inside the JSON string there is a JSON object literal
- This is a JSON string:

```
'{"fname":"Aine", "lname":"Dunne", "age":25}'
```

- This is a JSON object literal:

```
{"fname":"Aine", "lname":"Dunne", "age":25}
```

JSON vs XML

Both JSON and XML can be used to receive data from a web server.

The following JSON and XML examples both define an ***employees*** object, with an array of 3 ***employees***:

```
{"employees":[
    {"firstName":"Sean", "lastName":"Dunne"},
    {"firstName":"Aine", "lastName":"Carr"},
    {"firstName":"Paul", "lastName":"Quinn"}
  ]}
```

```
<employees>
  <employee>
    <firstName>Sean</firstName> <lastName>Dunne</lastName>
  </employee>
  <employee>
    <firstName>Aine</firstName> <lastName>Carr</lastName>
  </employee>
  <employee>
    <firstName>Paul</firstName> <lastName>Quinn</lastName>
  </employee>
</employees>
```

JSON is similar to 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 many programming languages

JSON is Different from XML Because

- JSON does not use end tags<>
- JSON is shorter
- JSON is quicker to read and write
- JSON can use arrays

- XML has to be parsed with an XML parser.
- JSON can be parsed by a standard function

person.json

person.json > ...

```json
{
  "firstName": "Peter",
  "lastName": "Pan",
  "hobbies": ["flying", "fencing", "staying young"],
  "age": 15,
  "friends": [
    {
      "firstName": "Tinkerbell",
      "age": 16
    },
    {
      "firstName": "Wendy",
      "age": 15
    }
  ]
}
```

- JSON supports primitive types, like strings and numbers, as well as nested lists and objects.
- It looks like a Python dictionary

Python Supports JSON

- Python comes with a built-in package called **json** for encoding and decoding JSON data.

```
import json
```

- The process of *encoding* JSON is often called *serialization*.
- This refers to the transformation of data into a series of bytes to be stored or transmitted across a network.
- *Deserialization* is the process of *decoding* data that has been stored or delivered in the JSON standard.
- Put simply, *encoding* is for *writing* data to disk, while *decoding* is for *reading* data into memory.

Serializing JSON

- **json** library methods `dump`() and `dumps`()
  - `dump`() method for writing data to files.
  - `dumps`() method (“dump-s”) for writing to a Python string.

Python objects are translated to JSON

Python	JSON
dict	object
list, tuple	array
str	string
int, long, float	number
True	true
False	false
None	null

A Simple Serialization Example

simpleWrite.py > ...

```python
import json

#python object
data = {
    "artist": {
        "name": "Bob Dylan",
        "Album": "Blood on the Tracks"
    }
}

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

- Creates a file called `artistFile.json` and opens it in write mode. (JSON files end in a *.json* extension.)

- `dump()` takes two arguments:
  1. the *data object* to be serialized
  2. the *file object* to which the bytes will be written.

Output: `artistFile.json`

{} artistFile.json > ...

```
{"artist": {"name": "Bob Dylan", "Album": "Blood on the Tracks"}}
```

`dumps()` method to write to a Python string

simpleDumps.py > ...

```python
import json

#python object
data = {
    "artist": {
        "name": "Bob Dylan",
        "Album": "Blood on the Tracks"
    }
}

#use dumps to create JSON string
myJsonString = json.dumps(data)
print(myJsonString)
```

Output:

```
[Running] python -u "c:\Users\Maeve.Carr\OneDrive - Atlantic TU\Se
{"artist": {"name": "Bob Dylan", "Album": "Blood on the Tracks"}}
```

Keyword Arguments - Kwargs

Both the `dump()` and `dumps()` methods can use the *keyword arguments*.
One example is `indent` keyword argument to specify indentation for nested structures

simpleWrite2Kwargs.py > ...

```python
import json

#python object
data = {
    "artist": {
        "name": "Bob Dylan",
        "Album": "Blood on the Tracks"
    }
}

with open("artistFile.json", "w") as writeFile:
    json.dump(data, writeFile, indent=4)


myJsonString = json.dumps(data)
print(myJsonString)
print()

myJsonFormatString = json.dumps(data, indent=4)
print(myJsonFormatString)
```

artistFile.json > ...

```json
{
    "artist": {
        "name": "Bob Dylan",
        "Album": "Blood on the Tracks"
    }
}
```

```
[Running] python -u "c:\Users\Maeve.Carr\OneDrive - Atlantic TU\Se
{"artist": {"name": "Bob Dylan", "Album": "Blood on the Tracks"}}

{
    "artist": {
        "name": "Bob Dylan",
        "Album": "Blood on the Tracks"
    }
}

[Done] exited with code=0 in 0.115 seconds
```

```
Type "help", "copyright", "credits" or "license()" for more information.
>>> import json

>>> data = {
    "artist": {
        "name": "Bob Dylan",
        "Album": "Blood on the Tracks"
    }
}
>>> json.dumps(data)
'{"artist": {"name": "Bob Dylan", "Album": "Blood on the Tracks"}}'
>>> json.dumps(data, indent=4)
'{\n    "artist": {\n        "name": "Bob Dylan",\n        "Album": "Blood on th
e Tracks"\n    }\n}'
>>>
```

Both the `dump`() and `dumps`() methods use the keyword arguments.

One example is indent keyword argument to specify indentation for nested structures

Deserializing JSON

- The ***json*** library, provides `load()` and `loads`() for converting JSON encoded data into Python objects.
- JSON to python object table for deserialization

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Deserializing JSON

- This conversion is not a perfect inverse to the serialization table.
- If you encode an object now and then decode it again later, you may not get exactly the same object back.
  - example encode a tuple and get back a list after decoding

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

A Simple Deserialization Example

deserializeEx.py > ...

```
1  import json
2
3  with open("artistFile.json", "r") as readFile:
4      data = json.load(readFile)
5
6  print(data)
7
```

- open the `artistFile.json` in *read* mode.
- Use `load()` to return python object
- Usually a `dict` or a `list`

```
Python\ReadingJSON\deserializeEx.py
{'artist': {'name': 'Bob Dylan', 'Album': 'Blood on the Tracks'}}
```

`json.load()` method

- The `json.load()` accepts the file object, parses the JSON data, populates a Python dictionary with the data, and returns it.

Syntax:

```
json.load(fileObject)
```

- ***Parameter***: It takes the file object as a parameter.
- ***Return***: It returns a JSON Object.

`json.loads()` Method (load-s)

- Parse a JSON string using the `json.loads()` method.
- `json.loads()` does not take the file path, but the file contents as a string.
- To read the content of a JSON file we can use `fileobject.read()` to convert the file into a string and pass it with `json.loads()`. This method returns the content of the file.

Syntax:

```
json.loads(S)
```

- ***Parameter***: it takes a string, bytes, or byte array instance which contains the JSON document as a parameter (S).

- ***Return Type***: It returns the Python object.

Using `loads()`

deserializeLoads.py > ...

```python
import json

# JSON string
jString = '{"name": "Seamus", "Nationality": "Irish"}'

# deserializes into dict and returns dict.
myDict = json.loads(jString)

print("JSON string = ", myDict)
print()

with open("artistFile.json", "r") as readFile:
    data = json.loads(readFile.read())

print(data)

```

```
[Running] python -u "c:\Users\Maeve.Carr\OneDrive - Atlantic TU\Se
JSON string =  {'name': 'Seamus', 'Nationality': 'Irish'}

{'artist': {'name': 'Bob Dylan', 'Album': 'Blood on the Tracks'}}

[Done] exited with code=0 in 0.111 seconds
```

- Shows reading from both string and JSON file using `json.loads()`.
- A JSON string is stored in a variable '`jString`' and converted to a Python dictionary `myDict` using `json.loads()`.
- Secondly, read JSON String stored in a file using `json.loads()`. Convert the JSON file into a string using the file handling and then convert it into the string using `read()` function using `json.loads()` method.

- If you need to parse a JSON string that returns a dictionary, then you can use the `json.loads()` method.

- If you need to parse a JSON file that returns a dictionary, then you can use the `json.load()` method.