**JSON**

JSON stands for Java Script Object Notation

JSON is a text file

JSON is format is used to exchange data in web application development.

JSON (JavaScript Object Notation, is an [open standard](https://en.wikipedia.org/wiki/Open_standard) file format and [data interchange](https://en.wikipedia.org/wiki/Electronic_data_interchange) format that uses [human-readable](https://en.wikipedia.org/wiki/Human-readable) text to store and transmit data objects consisting of [name–value pairs](https://en.wikipedia.org/wiki/Name%E2%80%93value_pair) and [arrays](https://en.wikipedia.org/wiki/Array_data_type) (or other [serializable](https://en.wikipedia.org/wiki/Serialization" \o "Serialization) values).

It is a commonly used data format with diverse uses in [electronic data interchange](https://en.wikipedia.org/wiki/Electronic_data_interchange), including that of [web applications](https://en.wikipedia.org/wiki/Web_application) with [servers](https://en.wikipedia.org/wiki/Server_(computing)).

JSON is a [language-independent](https://en.wikipedia.org/wiki/Language-independent_specification) data format. It was derived from [JavaScript](https://en.wikipedia.org/wiki/JavaScript), but many modern [programming languages](https://en.wikipedia.org/wiki/Programming_language) include code to generate and [parse](https://en.wikipedia.org/wiki/Parse) JSON-format data. JSON filenames use the extension .json

This json format understand by many programming languages which includes (java, python, .net, C, C++, javascript, PHP,..)

To work with json files python provides a module called “json”

“json” module provides encoders and decoders for converting python data types into json and json data into python

| **Python** | **JSON** |
| --- | --- |
| dict | object |
| list, tuple | array |
| str | string |
| int, float, int- & float-derived Enums | number |
| True | true |
| False | false |
| None | null |

json module provides the following methods

1. dump()
2. load()
3. dumps()
4. loads()

**dumps(obj)**

This function return json formatted string. This method does not allows to writing inside file

>>> import json

>>> emp\_dict={'empno':[1,2,3],

... 'ename':['naresh','ramesh','kishore'],

... 'sal':[65000,45000,55000]}

>>> print(emp\_dict)

{'empno': [1, 2, 3], 'ename': ['naresh', 'ramesh', 'kishore'], 'sal': [65000, 45000, 55000]}

>>> str1=json.dumps(emp\_dict)

>>> print(str1,type(str1))

{"empno": [1, 2, 3], "ename": ["naresh", "ramesh", "kishore"], "sal": [65000, 45000, 55000]} <class 'str'>

**dumps(obj,fobj)**

This function convert object into json string and write inside file

**Example:**

import json

emp\_dict={'empno':[1,2,3],

'ename':['naresh','ramesh','kishore'],

'sal':[65000,45000,55000]}

fobj=open("employee.json","w")

json.dump(emp\_dict,fobj)

fobj.close()

**Output**

Output is saved inside file

**loads()**

This function converts json string into python object type

>>> import json

>>> emp\_dict={'empno':[1,2,3],

... 'ename':['naresh','ramesh','kishore'],

... 'sal':[65000,45000,55000]}

>>> str1=json.dumps(emp\_dict)

>>> print(type(str1))

<class 'str'>

>>> emp\_dict1=json.loads(str1)

>>> print(type(emp\_dict1))

<class 'dict'>

>>> print(emp\_dict1)

{'empno': [1, 2, 3], 'ename': ['naresh', 'ramesh', 'kishore'], 'sal': [65000, 45000, 55000]}

>>>

**load()**

This function read json string from json file and return python object.

Syntax: load(fobj)

**Example**

import json

fobj=open("employee.json","r")

data=json.load(fobj)

print(data)

for key,value in data.items():

print(key,value)

**Output**

{'empno': [1, 2, 3], 'ename': ['naresh', 'ramesh', 'kishore'], 'sal': [65000, 45000, 55000]}

empno [1, 2, 3]

ename ['naresh', 'ramesh', 'kishore']

sal [65000, 45000, 55000]

**Binary files**

Binary file is a collection of bytes

Binary file allows only bytes data

To work with binary files, file must be open in

1. “wb” 🡪 write binary data
2. “ab” 🡪 append binary data
3. “rb” 🡪 read binary data

**Example:**

# Creating binary file

fobj=open("file1","wb")

b=bytes([65,66,67,68,69])

fobj.write(b)

fobj.close()

**Output**

Output is saved inside “file1”

**Example:**

# Reading binary file

fobj=open("file1","rb")

b=fobj.read()

print(type(b))

for value in b:

print(value)

fobj.close()

**Output**

<class 'bytes'>

65

66

67

68

69

**Example:**

# Write program to create copy of the image

fobj1=open("a1.jpg","rb")

fobj2=open("a2.jpg","wb")

b=fobj1.read(1024\*20)

fobj2.write(b)

print("file copied...")

fobj1.close()

fobj2.close()

**Output**

file copied...

**Pickle module or Object Serialization and De-Serialization**

The [pickle](file:///C:\Users\Satish%20Guptha%20Sir\AppData\Local\Programs\Python\Python312\Doc\html\library\pickle.html#module-pickle) module implements binary protocols for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream (from a [binary file](file:///C:\Users\Satish%20Guptha%20Sir\AppData\Local\Programs\Python\Python312\Doc\html\glossary.html#term-binary-file) or [bytes-like object](file:///C:\Users\Satish%20Guptha%20Sir\AppData\Local\Programs\Python\Python312\Doc\html\glossary.html#term-bytes-like-object)) is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,”  or “flattening”; however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

Pickle module provides the following functions for pickling and unpickling.

1. dump()
2. load()
3. dumps()
4. loads()

**Example:**

import pickle

a="PYTHON"

b=1.5

c=1+2j

d=[10,20,30]

e={1:10,2:20,3:30}

b1=pickle.dumps(a)

b2=pickle.dumps(b)

b3=pickle.dumps(c)

b4=pickle.dumps(d)

b5=pickle.dumps(e)

print(b1,b2,b3,b4,b5,sep="\n")

s1=pickle.loads(b1)

f1=pickle.loads(b2)

c1=pickle.loads(b3)

l1=pickle.loads(b4)

d1=pickle.loads(b5)

print(s1,f1,c1,l1,d1,sep="\n")

**Output**

b'\x80\x04\x95\n\x00\x00\x00\x00\x00\x00\x00\x8c\x06PYTHON\x94.'

b'\x80\x04\x95\n\x00\x00\x00\x00\x00\x00\x00G?\xf8\x00\x00\x00\x00\x00\x00.'

b'\x80\x04\x95.\x00\x00\x00\x00\x00\x00\x00\x8c\x08builtins\x94\x8c\x07complex\x94\x93\x94G?\xf0\x00\x00\x00\x00\x00\x00G@\x00\x00\x00\x00\x00\x00\x00\x86\x94R\x94.'

b'\x80\x04\x95\x0b\x00\x00\x00\x00\x00\x00\x00]\x94(K\nK\x14K\x1ee.'

b'\x80\x04\x95\x11\x00\x00\x00\x00\x00\x00\x00}\x94(K\x01K\nK\x02K\x14K\x03K\x1eu.'

PYTHON

1.5

(1+2j)

[10, 20, 30]

{1: 10, 2: 20, 3: 30}

**Example of writing inside file**

import pickle

a=65

b=1.5

c=1+2j

fobj=open("file2.ser","wb")

pickle.dump(a,fobj)

pickle.dump(b,fobj)

pickle.dump(c,fobj)

fobj.close()

**Output**

Output is saved inside file

**Example of reading bytes from file (de-serialization or unpickling)**

import pickle

fobj=open("file2.ser","rb")

a=pickle.load(fobj)

b=pickle.load(fobj)

c=pickle.load(fobj)

print(a,b,c)

**Output**

65 1.5 (1+2j)

|  |
| --- |
| **emp.py** |
| class Employee:  def \_\_int\_\_(self):  self.\_\_empno=None  self.\_\_ename=None  self.\_\_salary=None  def set\_data(self,eno,en,s):  self.\_\_empno=eno  self.\_\_ename=en  self.\_\_salary=s  def print\_data(self):  print(f'{self.\_\_empno},{self.\_\_ename},{self.\_\_salary}') |
| **Filetest1.py** |
| # Serialization  import emp  import pickle  emp1=emp.Employee()  emp2=emp.Employee()  emp3=emp.Employee()  emp1.set\_data(101,"naresh",45000)  emp2.set\_data(102,"suresh",34000)  emp3.set\_data(103,"kishore",25000)  fobj=open("employee.ser","wb")  pickle.dump(emp1,fobj)  pickle.dump(emp2,fobj)  pickle.dump(emp3,fobj)  fobj.close()  **Output**  Output is saved inside file |
| **Filetest2.py** |
| # De-Serialization  import pickle  fobj=open("employee.ser","rb")  emp1=pickle.load(fobj)  emp2=pickle.load(fobj)  emp3=pickle.load(fobj)  emp1.print\_data()  emp2.print\_data()  emp3.print\_data()  **Output**  101,naresh,45000  102,suresh,34000  103,kishore,25000 |