Persistence with Files in Python using JSON

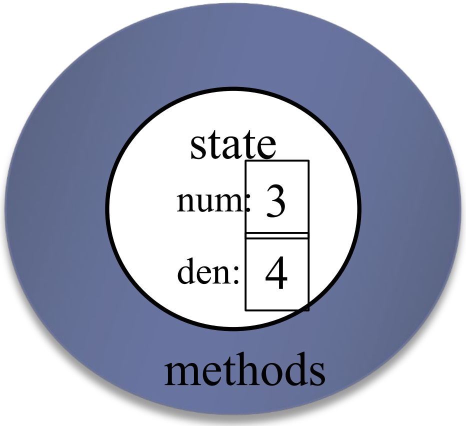
Roberto A. Bittencourt

JSON

Question?

- Given a particular set of data, how do you store it permanently?
  - What do you store on disk?
  - What format?
  - Can you easily transmit over the web?
  - Will it be readable by other languages?
  - Can humans read the data?

- Examples:
  - A square
  - A dictionary



Storage using plain text

- Advantages
  - Human readable (good for debugging / manual editing)
  - Portable to different platforms
  - Easy to transmit using web

- Disadvantages
  - Takes more memory than needed

- Use a standardized format – JSON
  - Makes the information more portable

JavaScript Object Notation

- Text-based notation for data interchange
  - Human readable

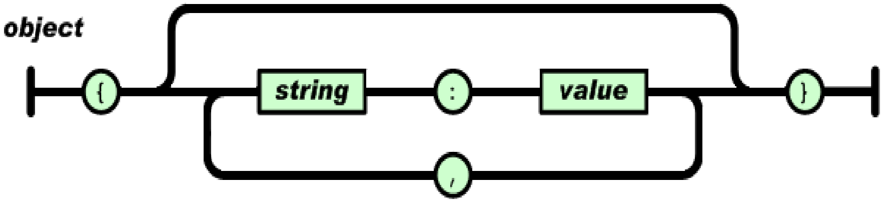
- Object
  - Unordered set of name-value pairs
  - names must be strings
  - `{ name1 : value1, name2 : value2, …, nameN : valueN }`

- Array
  - Ordered list of values
  - [value1, value2, … valueN]

JSON Data – A name and a value

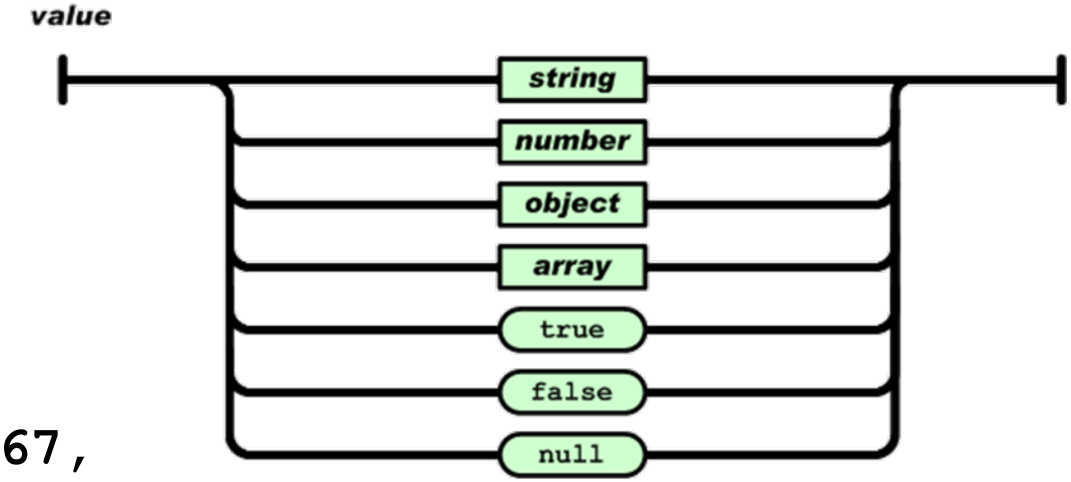
- A name/value pair consists of:
  - a field name (in **double quotes**), followed by a colon, followed by a value
- Unordered sets of name/value pairs
- Begins with **{** (left brace)
- Ends with **}** (right brace)
- Each name is followed by **:** (colon)
- Name/value pairs are separated by **,** (comma)

```
{
  "employee_id": 1234567,
  "name": "Jeff Fox",
  "hire_date": "1/1/2013",
  "location": "Norwalk, CT",
  "consultant": false
}
```



JSON Data – A name and a value

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



```
{
  "employee_id": 1234567,
  "name": "Jeff Fox",
  "hire_date": "1/1/2013",
  "location": "Norwalk, CT",
  "consultant": false
}
```

JSON Data – A name and a value

- Strings in JSON must be written in double quotes.

```
{ "name":"John" }
```

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

```
{ "age":30 }
```

- Values in JSON can be objects.

```
{
  "employee":{ "name":"John", "age":30, "city":"New York" }
}
```

- Values in JSON can be arrays.

```
{
  "employees":[ "John", "Anna", "Peter" ]
}
```

JSON basics in Python

Using JSON with Python

- To work with JSON (string or file containing JSON objects), you can use Python's JSON module.

```
import json
```

Loading JSON data from a file

- Example:

```
def load_json(filename):
    with open(filename, 'r') as file:
        jsn = json.load(file)
        #file.close()
    return jsn
person = load_json('person.json')
```

- This function above parses the **`person.json`** using **`json.load()`** method from the **`json`** module.
    - The result is a Python dictionary

Writing a JSON object into a file

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}

with open('person_to_json.json', 'w') as fp:
    json.dump(person, fp, indent=4)
```

- Using `json.dump()`, we can convert Python Objects into a JSON file.

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your `person.json` as follows.

```
person = load_json('person.json')
```

To access the "name" property:

```
  print(person["name"])
  John Smith
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your `person.json` as follows.

```
person = load_json('person.json')
```

To access the "age" property:

```
person["age"]
35
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your `person.json` as follows.

```
person = load_json('person.json')
```

To access the "street" property inside "address":

```
print(person["address"]["street"])
5 Main St.
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your `person.json` as follows.

```
person = load_json('person.json')
```

To access the "city" property inside "address":

```
print(person["address"]["city"])
Austin
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

Assume that you have already loaded your `person.json` as follows.

```
person = load_json('person.json')
```

To access the element at index 0 from the "children" property:

```
print(person["children"][0])
Mary
```

Accessing JSON Properties in Python

- Example:

```
person = { "name": "John Smith", "age": 35,
"address": {"street": "5 Main St.", "city":
"Austin"}, "children": ["Mary", "Abel"]}
```

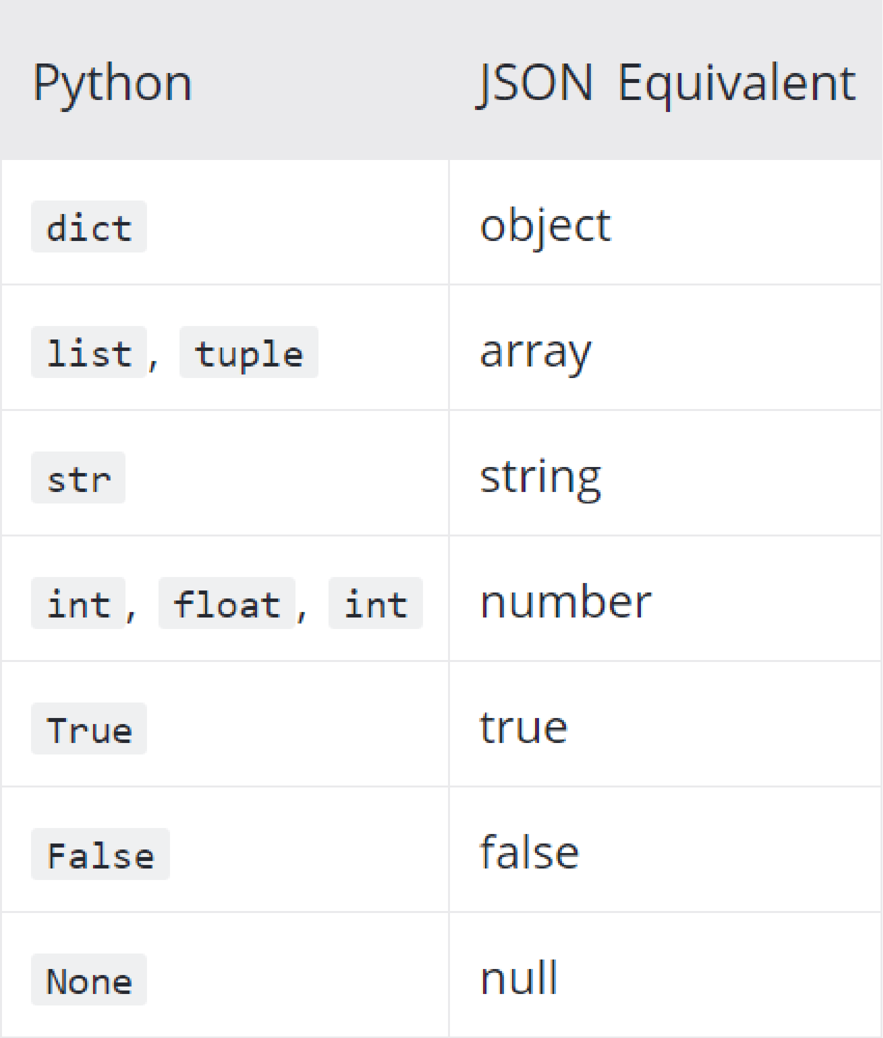
Assume that you have already loaded your `person.json` as follows.

```
person = load_json('person.json')
```

To access the element at index 1 from the "children" property:

```
print(person["children"][1])
Abel
```

Python – JSON Objects



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

More JSON File Handling in Python

Example01.py

Writing JSON using Python

- **`json.dumps( data )`**
  - Accepts Python object as an argument
  - Returns a string containing the information in JSON format
  - One typically write this string into a file
  - This operation is usually called serialization

```
def write(data, filename):
    file = open(filename, 'w')
    str_out = json.dumps(data)
    file.write(str_out)
    file.close()
```

`Example01.py`

Reading JSON using Python

- **`json.loads( data )`**
  - Accepts string as an argument
  - The string should be in JSON format
  - Returns a Python object corresponding to the data
  - This operation is usually called deserialization

Double quotes

```
"Hello World"
```

'hello.txt'

```
def read(filename):
    file = open(filename)
    str_in = file.read()
    file.close()
    data = json.loads(str_in)
    return data
```

```
write('Hello World', 'hello.txt')
print(read('hello.txt'))
```

`Example02.py`

Example 2: Writing a dictionary

- Create a dictionary

```
my_dict = {'Angela':'86620','adriana':'87113, 'ann':'84947'}
file_name = 'test_dict.txt'
write(my_dict, file_name)
```

```
{"ann": "84947", "adriana": "87113", "Angela": "86620"}
```

```
print(read(file_name))
```

Example03.py

Writing JSON using pretty printing

- **json.dumps(data)**

A dictionary

```
{'b': ['HELLO', 'WORLD'], 'a': ['hello', 'world']}
```

- **json.dumps**(data, indent=4, sort_keys=True)
  - Formats the output over multiple lines

```
{
    "a": [
        "hello",
        "world"
    ],
    "b": [
        "HELLO",
        "WORLD"
    ]
}
```

Double quotes

What about user-defined classes?

- Point class

```
class Point:
    def __init__(self, loc_x, loc_y):
        self.x = loc_x
        self.y = loc_y

    def __str__(self):
        return str(self.x) + ',' + str(self.y)
```

- If you can create a dictionary to store state information, then use JSON

value of x

value of y

```
p = Point(2, 3)
my_dict = {'__class__': 'Point', 'x' : p.x, 'y' : p.y}
```

What about user-defined classes?

- One can use JSON to read and extract the state information

```
file_name = 'test_point.txt'
write(my_dict, file_name)
```

```
{
    "__class__": "Point",
    "x": 2,
    "y": 3
}
```

- Example:

```
data = read(file_name)
result = Point( data['x'], data['y'] )
print (result)
```

JSON Encoding and Decoding

- Of course, manually creating dictionaries from objects and vice-versa is time-consuming and error-prone

- We may fix that by asking the **`json`** library to encode and decode objects through extending the classes:
  - **`json.JSONEncoder`**
  - **`json.JSONDecoder`**
- Then we call the **`encode()`** and **`decode()`** methods from the extended classes
- Let us look at an example in the following

Let us work with the Car class…

- Example:

```
class Car:
  def __init__(self, make, model, year, price):
    self.make = make
    self.model = model
    self.year = year
    self.price = price
```

- To create a new Car object, we can simply call the Car constructor with the appropriate arguments.

```
car = Car("Toyota", "Camry", 2022, 25000)
```

- If we try to serialize the Car object as-is, we will get a TypeError:

```
car_str = json.dumps(car)
TypeError: Object of type 'Car' is not JSON
serializable
```

Encoding the Car class…

- CarEncoder extends the JSONEncoder class:

```
class CarEncoder(json.JSONEncoder):
  def default(self, obj):
    if isinstance(obj, Car):
      return {"__type__": "Car", "make": obj.make, "model": \
       obj.model, "year": obj.year, "price": obj.price}
    return super().default(obj)
```

- Now we can get any Car object, encode it and save it.

```
car = Car("Toyota", "Camry", 2022, 25000)
car_json = json.dumps(car, cls=CarEncoder)
file.write(car_json)
print(car_json)
{"__type__": "Car", "make": "Toyota", "model": "Camry",
"year": 2022, "price": 25000}
```

Decoding the Car class…

- CarDecoder extends the JSONDecoder class:

```
class CarDecoder(json.JSONDecoder):
  def __init__(self, *args, **kwargs):
    super().__init__(object_hook=self.object_hook, *args, **kwargs)

  def object_hook(self, dct):
    if '__type__' in dct and dct['__type__'] == 'Car':
      return Car(dct['make'], dct['model'], dct['year'], dct['price'])
    return dct
```

- Now we can load any JSON dictionary representing a Car, decode it and create the Car object.

```
car_json = '{"__type__": "Car", "make": "Toyota", "model": "Camry",
"year": 2022, "price": 25000}'  # or use car_json = file.read()
car = json.loads(car_json, cls=CarDecoder)
print(car.make)   # Output: "Toyota"
print(car.model)  # Output: "Camry"
print(car.year)   # Output: 2022
print(car.price)  # Output: 25000
```