FEU Institute of Technology

(INTEGRATIVE PROGRAMMING AND TECHNOLOGIES)

EXERCISE

10

(Process JSON)

Name of Student	Name of Professor
Guillermo, Justine Rome	
Data Performed	Date Submitted

A (Very) Brief History of JSON

Not so surprisingly, JavaScript Object Notation was inspired by a subset of the JavaScript programming language dealing with object literal syntax. They've got a nifty website that explains the whole thing. Don't worry though: JSON has long since become language agnostic and exists as its own standard, so we can thankfully avoid JavaScript for the sake of this discussion.

Ultimately, the community at large adopted JSON because it's easy for both humans and machines to create and understand.

```JSON
{
    "firstName": "Jane",
    "lastName": "Doe",
    "hobbies": ["running", "sky diving", "singing"],
    "age": 35,
    "children": [
        {
            "firstName": "Alice",
            "age": 6
        },
        {
            "firstName": "Bob",
            "age": 8
        }
    ]
}
```

Python Supports JSON Natively!

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

Just throw this little guy up at the top of your file:

```python
import json
```

A Little Vocabulary

The process of encoding JSON is usually called **serialization**. This term refers to the transformation of data into a *series of bytes* (hence *serial*) to be stored or transmitted across a network. You may also hear the term **marshaling**

Naturally, **deserialization** is the reciprocal process of decoding data that has been stored or delivered in the JSON standard.

Serializing JSON

What happens after a computer processes lots of information? It needs to take a data dump. Accordingly, the `json` library exposes the `dump()` method for writing data to files. There is also a `dumps()` method (pronounced as "dump-s") for writing to a Python string.

Simple Python objects are translated to JSON according to a fairly intuitive conversion.

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

A Simple Serialization Example

Python

```
data = {
    "president": {
        "name": "Zaphod Beeblebrox",
        "species": "Betelgeusian"
    }
}
```

Python

```
with open("data_file.json", "w") as write_file:
    json.dump(data, write_file)
```

Python

```
json_string = json.dumps(data)
```

Python

```
>>> json.dumps(data)
>>> json.dumps(data, indent=4)
```

Deserializing JSON

Great, looks like you've captured yourself some wild JSON! Now it's time to whip it into shape. In the `json` library, you'll find `load()` and `loads()` for turning JSON encoded data into Python objects.

Just like serialization, there is a simple conversion table for deserialization, though you can probably guess what it looks like already.

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

Technically, this conversion isn't a perfect inverse to the serialization table. That basically means that if you encode an object now and then decode it again later, you may not get exactly the same object back. I imagine it's a bit like teleportation: break my molecules down over here and put them back together over there. Am I still the same person,

In reality, it's probably more like getting one friend to translate something into Japanese and another friend to translate it back into English. Regardless, the

simplest example would be encoding a `tuple` and getting back a `list` after decoding, like so:

Python

```
>>> blackjack_hand = (8, "Q")
>>> encoded_hand = json.dumps(blackjack_hand)
>>> decoded_hand = json.loads(encoded_hand)

>>> blackjack_hand == decoded_hand
False
>>> type(blackjack_hand)
<class 'tuple'>
>>> type(decoded_hand)
<class 'list'>
>>> blackjack_hand == tuple(decoded_hand)
True
```

A Simple Deserialization Example

This time, imagine you've got some data stored on disk that you'd like to manipulate in memory. You'll still use the context manager, but this time you'll open up the existing `data_file.json` in read mode.

Python

```
with open("data_file.json", "r") as read_file:
    data = json.load(read_file)
```

Things are pretty straightforward here, but keep in mind that the result of this method could return any of the allowed data types from the conversion table. This is only important if you're loading in data you haven't seen before. In most cases, the root object will be a `dict` or a `list`.

If you've pulled JSON data in from another program or have otherwise obtained a string of JSON formatted data in Python, you can easily deserialize that with loads(), which naturally loads from a string:

Python

```
json_string = """
{
    "researcher": {
        "name": "Ford Prefect",
        "species": "Betelgeusian",
        "relatives": [
            {
                "name": "Zaphod Beeblebrox",
                "species": "Betelgeusian"
            }
        ]
    }
}
"""
data = json.loads(json_string)
```

]A Real World Example (sort of)

Python

```
import json
import requests
```

Python

```
response = requests.get("https://jsonplaceholder.typicode.com/todos")
todos = json.loads(response.text)
```

Python

```
>>> todos == response.json()
True
>>> type(todos)
<class 'list'>
>>> todos[:10]
...
```

JSON

```
{
    "userId": 1,
    "id": 1,
    "title": "delectus aut autem",
    "completed": false
}
```

Python

```python
# Map of userId to number of complete TODOs for that user
todos_by_user = {}

# Increment complete TODOs count for each user.
for todo in todos:
    if todo["completed"]:
        try:
            # Increment the existing user's count.
            todos_by_user[todo["userId"]] += 1
        except KeyError:
            # This user has not been seen. Set their count to 1.
            todos_by_user[todo["userId"]] = 1

# Create a sorted list of (userId, num_complete) pairs.
top_users = sorted(todos_by_user.items(),
                   key=lambda x: x[1], reverse=True)
```

```python
# Get the maximum number of complete TODOs.
max_complete = top_users[0][1]

# Create a list of all users who have completed
# the maximum number of TODOs.
users = []
for user, num_complete in top_users:
    if num_complete < max_complete:
        break
    users.append(str(user))

max_users = " and ".join(users)
```

Python

```
>>> s = "s" if len(users) > 1 else ""
>>> print(f"user{s} {max_users} completed {max_complete} TODOs")
users 5 and 10 completed 12 TODOs
```

Python

```
# Define a function to filter out completed TODOs
# of users with max completed TODOS.
def keep(todo):
    is_complete = todo["completed"]
    has_max_count = str(todo["userId"]) in users
    return is_complete and has_max_count

# Write filtered TODOs to file.
with open("filtered_data_file.json", "w") as data_file:
    filtered_todos = list(filter(keep, todos))
    json.dump(filtered_todos, data_file, indent=2)
```

Code

```
import json
import requests

response = requests.get('https://jsonplaceholder.typicode.com/todos/')

todos = json.loads(response.text)

todos_by_user = {}

for todo in todos:
    if todo["completed"]:
        try:
            todos_by_user[todo["userId"]]+=1
        except KeyError:
            todos_by_user[todo["userId"]]=1
top_users = sorted(todos_by_user.items(),
                   key=lambda x: x[1], reverse=True)

max_complete = top_users[0][1]
users = []
for user, num_complete in top_users:
    if num_complete < max_complete:
        break
    users.append(str(user))
max_users = "and".join(users)

def keep(todo):
    is_complete = todo["completed"]
    has_max_count = str(todo["userId"]) in users
    return is_complete and has_max_count
with open("C:/Users/Justine/Desktop/filtered_data_file.json","w")as data_file:
    filtered_todos = list(filter(keep, todos))
    json.dump(filtered_todos, data_file, indent=2)
    print('all done')
```

Filtered JSON file


```
[
  {
    "userId": 5,
    "id": 81,
    "title": "suscipit qui totam",
    "completed": true
  },
  {
    "userId": 5,
    "id": 83,
    "title": "quidem at rerum quis ex aut sit quam",
    "completed": true
  },
  {
    "userId": 5,
    "id": 85,
    "title": "et quia ad iste a",
    "completed": true
  },
  {
    "userId": 5,
    "id": 86,
    "title": "incidunt ut saepe autem",
    "completed": true
  },
  {
    "userId": 5,
    "id": 87,
    "title": "laudantium quae eligendi consequatur quia et vero autem",
    "completed": true
  },
  {
    "userId": 5,
    "id": 89,
    "title": "sequi ut omnis et",
    "completed": true
  },
  {
    "userId": 5,
    "id": 90,
    "title": "molestiae nisi accusantium tenetur dolorem et",
    "completed": true
  },
```

```
  {
    "userId": 5,
    "id": 91,
    "title": "nulla quis consequatur saepe qui id expedita",
    "completed": true
  },
  {
    "userId": 5,
    "id": 92,
    "title": "in omnis laboriosam",
    "completed": true
  },
  {
    "userId": 5,
    "id": 93,
    "title": "odio iure consequatur molestiae quibusdam necessitatibus quia sint"
,
    "completed": true
  },
  {
    "userId": 5,
    "id": 95,
    "title": "vel nihil et molestiae iusto assumenda nemo quo ut",
    "completed": true
  },
  {
    "userId": 5,
    "id": 98,
    "title": "debitis accusantium ut quo facilis nihil quis sapiente necessitatib
us",
    "completed": true
  },
  {
    "userId": 10,
    "id": 182,
    "title": "inventore saepe cumque et aut illum enim",
    "completed": true
  },
  {
    "userId": 10,
    "id": 183,
    "title": "omnis nulla eum aliquam distinctio",
    "completed": true
  },
  {
```

```
    "userId": 10,
    "id": 188,
    "title": "vel non beatae est",
    "completed": true
  },
  {
    "userId": 10,
    "id": 189,
    "title": "culpa eius et voluptatem et",
    "completed": true
  },
  {
    "userId": 10,
    "id": 190,
    "title": "accusamus sint iusto et voluptatem exercitationem",
    "completed": true
  },
  {
    "userId": 10,
    "id": 191,
    "title": "temporibus atque distinctio omnis eius impedit tempore molestias pa
riatur",
    "completed": true
  },
  {
    "userId": 10,
    "id": 193,
    "title": "rerum debitis voluptatem qui eveniet tempora distinctio a",
    "completed": true
  },
  {
    "userId": 10,
    "id": 195,
    "title": "rerum ex veniam mollitia voluptatibus pariatur",
    "completed": true
  },
  {
    "userId": 10,
    "id": 196,
    "title": "consequuntur aut ut fugit similique",
    "completed": true
  },
  {
    "userId": 10,
    "id": 197,
```

```
    "title": "dignissimos quo nobis earum saepe",
    "completed": true
  },
  {
    "userId": 10,
    "id": 198,
    "title": "quis eius est sint explicabo",
    "completed": true
  },
  {
    "userId": 10,
    "id": 199,
    "title": "numquam repellendus a magnam",
    "completed": true
  }
]
```