Loading Data from a JSON File

Like the `pickle` module, the `json` module has a `load()` function which takes a stream object, reads JSON-encoded data from it, and creates a new Python object that mirrors the JSON data structure.

```
shell = 2
print (shell)
#2

del entry                                         #①
print (entry)
#Traceback (most recent call last):
#  File "<stdin>", line 1, in <module>
#NameError: name 'entry' is not defined
```

```
import json
with open('entry.json', 'r', encoding='utf-8') as f:
     entry = json.load(f)                         #②

print (entry)                                     #③
#{'comments_link': None,
# 'internal_id': {'__class__': 'bytes', '__value__': [222, 213, 180, 248]},
# 'title': 'Dive into history, 2009 edition',
# 'tags': ['diveintopython', 'docbook', 'html'],
# 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-editio
# 'published_date': {'__class__': 'time.asctime', '__value__': 'Fri Mar 27 22:20:42 2009'},
# 'published': True}
```

① For demonstration purposes, switch to Python Shell #2 and delete the `entry` data structure that you created earlier in this chapter with the `pickle` module.

② In the simplest case, the `json.load()` function works the same as the `pickle.load()` function. You pass in a stream object and it returns a new Python object.

③ I have good news and bad news. Good news first: the `json.load()` function successfully read the `entry.json` file you created in Python Shell #1 and

created a new Python object that contained the data. Now the bad news: it didn't recreate the original `entry` data structure. The two values `'internal_id'` and `'published_date'` were recreated as dictionaries — specifically, the dictionaries with JSON-compatible values that you created in the `to_json()` conversion function.

`json.load()` doesn't know anything about any conversion function you may have passed to `json.dump()`. What you need is the opposite of the `to_json()` function — a function that will take a custom-converted JSON object and convert it back to the original Python datatype.

```
# add this to customserializer.py
def from_json(json_object):                                   #①
    if '__class__' in json_object:                            #②
        if json_object['__class__'] == 'time.asctime':
            return time.strptime(json_object['__value__'])    #③
        if json_object['__class__'] == 'bytes':
            return bytes(json_object['__value__'])            #④
    return json_object
```

① This conversion function also takes one parameter and returns one value. But the parameter it takes is not a string, it's a Python object — the result of deserializing a JSON-encoded string into Python.

② All you need to do is check whether this object contains the `'__class__'` key that the `to_json()` function created. If so, the value of the `'__class__'` key will tell you how to decode the value back into the original Python datatype.

③ To decode the time string returned by the `time.asctime()` function, you use the `time.strptime()` function. This function takes a formatted datetime string (in a customizable format, but it defaults to the same format that `time.asctime()` defaults to) and returns a `time.struct_time`.

④ To convert a list of integers back into a `bytes` object, you can use the `bytes()` function.

That was it; there were only two datatypes handled in the `to_json()` function, and now those two datatypes are handled in the `from_json()` function. This is the result:

```
shell = 2
print (shell)

#2

import customserializer
with open('entry.json', 'r', encoding='utf-8') as f:
     entry = json.load(f, object_hook=customserializer.from_json) #①

print (entry )                                                     #②
#{'comments_link': None,
# 'internal_id': b'\xDE\xD5\xB4\xF8',
# 'title': 'Dive into history, 2009 edition',
# 'tags': ['diveintopython', 'docbook', 'html'],
# 'article_link': 'http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-editior
# 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=2
# 'published': True}
```

① To hook the `from_json()` function into the deserialization process, pass it as the `object_hook` parameter to the `json.load()` function. Functions that take functions; it's so handy!

② The `entry` data structure now contains an `'internal_id'` key whose value is a `bytes` object. It also contains a `'published_date'` key whose value is a `time.struct_time object`.

There is one final glitch, though.

```
shell = 1
print (shell)
#1

import customserializer
with open('entry.json', 'r', encoding='utf-8') as f:
     entry2 = json.load(f, object_hook=customserializer.from_json)

print (entry2 == entry)                                                 #①
#False

print (entry['tags'] )                                                  #②
#('diveintopython', 'docbook', 'html')

print (entry2['tags'])                                                  #③
#['diveintopython', 'docbook', 'html']
```

① Even after hooking the `to_json()` function into the serialization, and hooking the `from_json()` function into the deserialization, we still haven't

recreated a perfect replica of the original data structure. Why not?

② In the original `entry` data structure, the value of the `'tags'` key was a tuple of three strings.

③ But in the round-tripped `entry2` data structure, the value of the `'tags'` key is a *list* of three strings. JSON doesn't distinguish between tuples and lists; it only has a single list-like datatype, the array, and the `json` module silently converts both tuples and lists into JSON arrays during serialization. For most uses, you can ignore the difference between tuples and lists, but it's something to keep in mind as you work with the `json` module.