Serializing Datatypes Unsupported by JSON

Even if JSON has no built-in support for bytes, that doesn't mean you can't serialize `bytes` objects. The `json` module provides extensibility hooks for encoding and decoding unknown datatypes. (By "unknown," I mean "not defined in JSON." Obviously the `json` module knows about byte arrays, but it's constrained by the limitations of the JSON specification.) If you want to encode bytes or other datatypes that JSON doesn't support natively, you need to provide custom encoders and decoders for those types.

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

print (entry)                                                    #①
#{'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-edition
# 'published_date': time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=2
# 'published': True}
```

```
import json
with open('entry.json', 'w', encoding='utf-8') as f:  #②
    json.dump(entry, f)                               #③

#Traceback (most recent call last):
#  File "<stdin>", line 5, in <module>
#  File "C:\Python31\lib\json\__init__.py", line 178, in dump
#    for chunk in iterable:
#  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
#    for chunk in _iterencode_dict(o, _current_indent_level):
#  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
#    for chunk in chunks:
#  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
#    o = _default(o)
#  File "C:\Python31\lib\json\encoder.py", line 170, in default
#    raise TypeError(repr(o) + " is not JSON serializable")
#TypeError: b'\xDE\xD5\xB4\xF8' is not JSON serializable
```

① OK, it's time to revisit the `entry` data structure. This has it all: a boolean value, a `None` value, a string, a tuple of strings, a `bytes` object, and a `time` structure.

② I know I've said it before, but it's worth repeating: JSON is a text-based format. Always open JSON files in text mode with a UTF-8 character encoding.

③ Well *that's* not good. What happened?

Here's what happened: the `json.dump()` function tried to serialize the `bytes` object `b'\xDE\xD5\xB4\xF8'`, but it failed, because JSON has no support for `bytes` objects. However, if storing bytes is important to you, you can define your own "mini-serialization format."

```
def to_json(python_object):                                    #①
    if isinstance(python_object, bytes):                       #②
        return {'__class__': 'bytes',
                '__value__': list(python_object)}              #③
    raise TypeError(repr(python_object) + ' is not JSON serializable')  #④
```

① To define your own "mini-serialization format" for a datatype that json doesn't support natively, just define a function that takes a Python object as a parameter. This Python object will be the actual object that the `json.dump()` function is unable to serialize by itself — in this case, the `bytes` object `b'\xDE\xD5\xB4\xF8'`. ② Your custom serialization function should check the type of the Python object that the `json.dump()` function passed to it. This is not strictly necessary if your function only serializes one datatype, but it makes it crystal clear what case your function is covering, and it makes it easier to extend if you need to add serializations for more datatypes later.

③ In this case, I've chosen to convert a bytes object into a dictionary. The `__class__` key will hold the original datatype (as a string, `'bytes'`), and the `__value__` key will hold the actual value. Of course this can't be a `bytes` object; the entire point is to convert it into something that can be serialized in JSON! A `bytes` object is just a sequence of integers; each integer is somewhere in the range 0–255. We can use the `list()` function to convert the bytes object into a list of integers. So `b'\xDE\xD5\xB4\xF8'` becomes `[222, 213, 180, 248]`.

(Do the math! It works! The byte `\xDE` in hexadecimal is 222 in decimal, `\xD5` is 213, and so on.)

④ This line is important. The data structure you're serializing may contain types that neither the built-in JSON serializer nor your custom serializer can handle. In this case, your custom serializer must raise a `TypeError` so that the `json.dump()` function knows that your custom serializer did not recognize the type.

That's it; you don't need to do anything else. In particular, this custom serialization function *returns a Python dictionary*, not a string. You're not doing the entire serializing-to-JSON yourself; you're only doing the converting-to-a-supported-datatype part. The `json.dump()` function will do the rest.

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

import customserializer                                                    #①
with open('entry.json', 'w', encoding='utf-8') as f:                       #②
    json.dump(entry, f, default=customserializer.to_json)                  #③

#Traceback (most recent call last):
#  File "<stdin>", line 9, in <module>
#    json.dump(entry, f, default=customserializer.to_json)
#  File "C:\Python31\lib\json\__init__.py", line 178, in dump
#    for chunk in iterable:
#  File "C:\Python31\lib\json\encoder.py", line 408, in _iterencode
#    for chunk in _iterencode_dict(o, _current_indent_level):
#  File "C:\Python31\lib\json\encoder.py", line 382, in _iterencode_dict
#    for chunk in chunks:
#  File "C:\Python31\lib\json\encoder.py", line 416, in _iterencode
#    o = _default(o)
#  File "/Users/pilgrim/diveintopython3/examples/customserializer.py", line 12, in to_json
    raise TypeError(repr(python_object) + ' is not JSON serializable')     ④
TypeError: time.struct_time(tm_year=2009, tm_mon=3, tm_mday=27, tm_hour=22, tm_min=20, tm_sec
```

① The `customserializer` module is where you just defined the `to_json()` function in the previous example.

② Text mode, UTF-8 encoding, yadda yadda. (You'll forget! I forget sometimes! And everything will work right up until the moment that it fails, and then it will fail most spectacularly.)

③ This is the important bit: to hook your custom conversion function into the `json.dump()` function, pass your function into the `json.dump()` function in the `default` parameter. (Hooray, everything in Python is an object!)

④ OK, so it didn't actually work. But take a look at the exception. The `json.dump()` function is no longer complaining about being unable to serialize the `bytes` object. Now it's complaining about a completely different object: the `time.struct_time` object.

While getting a different exception might not seem like progress, it really is! It'll just take one more tweak to get past this.

```
import time

def to_json(python_object):
    if isinstance(python_object, time.struct_time):          #①
        return {'__class__': 'time.asctime',
                '__value__': time.asctime(python_object)}    #②
    if isinstance(python_object, bytes):
        return {'__class__': 'bytes',
                '__value__': list(python_object)}
    raise TypeError(repr(python_object) + ' is not JSON serializable')
```

① Adding to our existing `customserializer.to_json()` function, we need to check whether the Python object (that the `json.dump()` function is having trouble with) is a `time.struct_time`.

② If so, we'll do something similar to the conversion we did with the `bytes` object: convert the `time.struct_time` object to a dictionary that only contains JSON-serializable values. In this case, the easiest way to convert a datetime into a JSON-serializable value is to convert it to a string with the `time.asctime()` function. The `time.asctime()` function will convert that nasty-looking `time.struct_time` into the string `'Fri Mar 27 22:20:42 2009'`.

With these two custom conversions, the entire `entry` data structure should serialize to json without any further problems.

```
shell = 1

with open('entry.json', 'w', encoding='utf-8') as f:
     json.dump(entry, f, default=customserializer.to_json)
```

```
you@localhost:~/diveintopython3/examples$ ls -l example.json
-rw-r--r-- 1 you  you  391 Aug  3 13:34 entry.json
you@localhost:~/diveintopython3/examples$ cat example.json
{"published_date": {"__class__": "time.asctime", "__value__": "Fri Mar 27 22:20:42 2009"},
"comments_link": null, "internal_id": {"__class__": "bytes", "__value__": [222, 213, 180, 248
"tags": ["diveintopython", "docbook", "html"], "title": "Dive into history, 2009 edition",
"article_link": "http://diveintomark.org/archives/2009/03/27/dive-into-history-2009-edition",
"published": true}
```

```
"published": true}
```