**Serialization techniques in Python**

[[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----297a4489ba87--------------------------------)](https://gaurav-adarshi.medium.com/?source=post_page-----297a4489ba87--------------------------------)

[Gaurav Kumar](https://gaurav-adarshi.medium.com/?source=post_page-----297a4489ba87--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2Ff319c9e06fc1&operation=register&redirect=https%3A%2F%2Fgaurav-adarshi.medium.com%2Fserialization-techniques-in-python-297a4489ba87&user=Gaurav+Kumar&userId=f319c9e06fc1&source=post_page-f319c9e06fc1----297a4489ba87---------------------post_header-----------)

12 min read

·

6 days ago

235

There are many situations where you would want to store or share complex data in a file. These are the situations where serialization comes in handy. In this article, we will learn about Python serialization and how it can be implemented using python module. Then, we will also see in brief how serialization and deserialization can be done using python module.

**What is Serialization in Python?**

Serialization in Python refers to the process of converting complex data structures, such as objects or data collections, into a format that can be easily stored or transmitted. This is often done using modules like pickle or JSON. The serialized data can later be deserialized to reconstruct the original data structure.

Python offers three modules for the purpose of data serialization:

1. Pickle Module
2. JSON Module
3. Marshal Module

This article primarily focuses on the Pickle module, which is the most straightforward method for storing intricate data in a specialized format.

**Pickle Module in Python**

The Pickle module in Python is used for object serialization. It allows you to convert complex Python objects into a byte stream, making it easy to store or transmit them.

**Pickle Module Overview:**

The pickle module serves the purpose of employing binary protocols to serialize and deserialize Python object structures.

**Pickling** involves the transformation of a Python object hierarchy into a byte stream.

**Unpickling**, on the other hand, is the reverse of the pickling process, where a byte stream is transformed back into an object hierarchy.

**Module Interface**

**dumps()** is a function utilized for the serialization of an object hierarchy, converting it into a byte stream.

**loads()** is the counterpart function responsible for the deserialization of a data stream, reconstructing the original object hierarchy.

**Constants provided by the pickle module**

**`pickle.HIGHEST\_PROTOCOL`** is an integer that signifies the highest available protocol version for pickling. It serves as the protocol parameter value when using functions like **`dump()`** and **`dumps()`**.

**`pickle.DEFAULT\_PROTOCOL`** is also an integer, representing the default pickling protocol. Its value is typically lower than the value of the highest protocol and is automatically used if a specific protocol is not explicitly provided.

**Functions provided by the pickle module**

The function **`pickle.dump(obj, file, protocol=None, \*, fix\_imports=True)`** performs the same operation as **`Pickler(file, protocol).dump(obj)`**. Its purpose is to write a pickled representation of the object `obj` to the specified open file object **`file`**.

The optional**`protocol`** argument, which is an integer, specifies the protocol to be used by the pickler. The supported protocols range from 0 to HIGHEST\_PROTOCOL. If not explicitly provided, the default is DEFAULT\_PROTOCOL. In case a negative number is provided, HIGHEST\_PROTOCOL is automatically selected.

The parameter **`fix\_imports`** comes into play when it is set to true, and the protocol is less than 3. In such cases, the pickle operation will attempt to map the new Python 3 names to the old module names used in Python 2. This is done to ensure that the pickle data stream remains readable with Python.

**Example:**The following code illustrates the definition of a SimpleObject class, the creation of instances stored in a list, and the use of the pickle module to serialize these instances. The resulting pickled data is then written to an io.StringIO object.

import pickle  
import io  
  
class SimpleObject(object):  
 def \_\_init\_\_(self, name):  
 self.name = name  
 l = list(name)  
 l.reverse()  
 self.name\_backwards = ''.join(l)  
 return  
  
data = []  
data.append(SimpleObject('pickle'))  
data.append(SimpleObject('cPickle'))  
data.append(SimpleObject('last'))  
  
out\_s = io.BytesIO() # Use BytesIO for binary data  
for o in data:  
 print('WRITING: %s (%s)' % (o.name, o.name\_backwards))  
 pickle.dump(o, out\_s)  
 out\_s.flush()

**Output:**  
WRITING: pickle (elkcip)  
WRITING: cPickle (elkciPc)  
WRITING: last (tsal)

The function **`pickle.dumps(obj, protocol=None, \*, fix\_imports=True)`** in Python produces the serialized representation of the provided object as a bytes object.

For instance, consider the following code utilizing the **‘pickle’** module to serialize a list that encompasses a dictionary containing diverse data types (such as string, integer, and float). It transforms the data into a binary string, and this output serves as the serialized data. This binary string can be utilized with **`pickle.loads()`** to reconstruct the initial data structure.

import pickle  
  
# Creating a sample data structure  
person\_info = {'name': 'John Doe', 'age': 30, 'country': 'USA'}  
  
# Serializing the data using pickle.dumps()  
serialized\_data = pickle.dumps(person\_info)  
  
# Printing the serialized data  
print('PICKLE:', serialized\_data)

**Output:**  
PICKLE: b’\x80\x04\x95/\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x04name\x94\x8c\x08John Doe\x94\x8c\x03age\x94K\x1e\x8c\x07country\x94\x8c\x03USA\x94u.’

The **pickle.load(file, \*, fix\_imports=True, encoding="ASCII", errors="strict")** function is synonymous with **Unpickler(file).load()**. It reads a pickled object representation from the open file object file and returns the reconstituted object hierarchy specified.

**Example:** The following code defines a class named **‘SimpleObject’** and creates instances stored in a list. It utilizes the pickle module to serialize these instances, writing them to an io.StringIO object. Subsequently, it reads and deserializes the pickled data from **in\_s** using **pickle.load()**. The code exemplifies the complete cycle of pickling and unpickling, successfully displaying information about the unpickled objects.

import pickle  
import io  
  
class SimpleObject(object):  
 def \_\_init\_\_(self, name):  
 self.name = name  
 l = list(name)  
 l.reverse()  
 self.name\_backwards = ''.join(l)  
 return  
  
data = []  
data.append(SimpleObject('pickle'))  
data.append(SimpleObject('cPickle'))  
data.append(SimpleObject('last'))  
  
out\_s = io.BytesIO()  
  
for obj in data:  
 print('WRITING: %s (%s)' % (obj.name, obj.name\_backwards))  
 pickle.dump(obj, out\_s)  
 out\_s.flush()  
  
# Reset the stream position to the beginning  
out\_s.seek(0)  
  
in\_s = io.BytesIO(out\_s.getvalue())  
  
while True:  
 try:  
 loaded\_obj = pickle.load(in\_s)  
 except EOFError:  
 break  
 else:  
 print('READ: %s (%s)' % (loaded\_obj.name, loaded\_obj.name\_backwards))

**Output:**  
WRITING: pickle (elkcip)  
WRITING: cPickle (elkciPc)  
WRITING: last (tsal)  
READ: pickle (elkcip)  
READ: cPickle (elkciPc)  
READ: last (tsal)

The **pickle.loads(bytes\_object, \*, fix\_imports=True, encoding="ASCII", errors="strict")** function is employed to retrieve a pickled object representation from a bytes object and return the reconstructed object hierarchy as specified.

Example: The following code illustrates the process of pickling and unpickling data using the pickle module. Initially, it serializes a list of dictionaries and subsequently loads the pickled data into another variable. The code then compares the original and unpickled data, demonstrating that they are equivalent in content **(data1 == data2)** but distinct objects in memory **(data1 is not data2)**. This showcases the essence of serialization and deserialization while preserving the integrity of the data.

import pickle  
import pprint  
  
data1 = [{'a': 'A', 'b': 2, 'c': 3.0}]  
print('BEFORE:')  
pprint.pprint(data1)  
  
data1\_string = pickle.dumps(data1)  
  
data2 = pickle.loads(data1\_string)  
print('AFTER:')  
pprint.pprint(data2)  
  
print('SAME?:', (data1 is data2))  
print('EQUAL?:', (data1 == data2))

**Output:**  
BEFORE: [{‘a’: ‘A’, ‘b’: 2, ‘c’: 3.0}]  
AFTER: [{‘a’: ‘A’, ‘b’: 2, ‘c’: 3.0}]  
SAME?: False  
EQUAL?: True

In this example, the code uses the **pickle.dumps()** function to serialize the **data1** list of dictionaries into a byte stream **(data1\_string).** The **pickle.loads()** function is then utilized to deserialize the byte stream back into a Python object**(data2)**. The final print statements compare the identity **(is)** and equality **(==)**of the original and unpickled data, demonstrating the behavior of serialization and deserialization.

**Exceptions provided by the pickle module**

The **`pickle.PickleError`** exception serves as the root class for all other exceptions that arise during the pickling process.

The **`pickle.PicklingError`** exception, a subclass of **`PickleError`**, comes into play when a Pickler encounters an object that cannot be serialized, presenting a situation where pickling is not possible.

On the other hand, the **`pickle.UnpicklingError`** exception, also inheriting from **`PickleError`**, is raised when issues like data corruption or security breaches are encountered during the unpickling process. It indicates problems that may occur while reconstructing an object from a pickled byte stream.

**Classes exported by the pickle module:**

**class pickle.Pickler(file,protocol=None,\*,fix\_imports=True)**

This class takes a binary file for writing a pickle data stream.  
**dump(obj) —**This function is used to write a pickled representation of obj to the open file object given in the constructor.

**persistent\_id(obj) —**If persistent\_id() returns None, obj is pickled as usual. This does nothing by default and exists so that any subclass can override it.

**Dispatch\_table —**A pickler object’s dispatch table is a mapping whose keys are classes and whose values are reduction functions.  
By default, a pickler object will not have a dispatch\_table attribute, and it will instead use the global dispatch table managed by the copyreg module.  
**Example :**The below code creates an instance of pickle.Pickler with a private dispatch table that handles the SomeClass class especially.

import pickle  
import io  
import copyreg  
  
class SomeClass:  
 def \_\_init\_\_(self, data):  
 self.data = data  
  
# Function to reduce instances of SomeClass  
def reduce\_SomeClass(obj):  
 return (SomeClass, (obj.data,))  
  
data\_to\_serialize = SomeClass("example\_data")  
  
# Create a binary stream  
f = io.BytesIO()  
  
# Create a Pickler instance  
p = pickle.Pickler(f)  
  
# Set up a dispatch table  
p.dispatch\_table = copyreg.dispatch\_table.copy()  
  
p.dispatch\_table = {SomeClass: reduce\_SomeClass}

**class pickle.Unpickler(file,\*,fix\_imports=True,encoding="ACSII",errors="strict")**

This class takes a binary file for reading a pickle data stream.  
**load() —**This function is used to read a pickled object representation from the open file object file and return the reconstituted object hierarchy specified.

**persistent\_load(pid) —**This raises an UnpicklingError by default.

**find\_class(module, name) —**This function imports the module if required and returns the object called name from it, where the module and name arguments are str objects.

**Pickling Class Instances**

This segment outlines the fundamental mechanisms for defining, customizing, and managing the pickling and unpickling processes of class instances. Making instances picklable typically requires no extra code, as pickle automatically utilizes introspection to obtain the class and attributes of an instance.

However, classes have the option to modify this default behavior by incorporating specific methods. These special methods enable classes to exert control over the pickling and unpickling procedures.

**object.\_\_getnewargs\_ex\_\_():**

The **`object.\_\_getnewargs\_ex\_\_()`** method governs the parameters supplied to the **`\_\_new\_\_()`** method during unpickling. It is required to return a tuple **`(args, kwargs)`**, where **`args`** is a sequence of positional arguments and **`kwargs`** is a dictionary containing named arguments for constructing the object. This method provides a way to customize how instances are recreated during the unpickling process.

**object.\_\_getnewargs\_\_():**

The method **`object.\_\_getnewargs\_\_()`** exclusively handles positive arguments and is designed to return a tuple of arguments **(`args`)**. These arguments are then passed to the **`\_\_new\_\_()`** method during the unpickling process.

**object.\_\_getstate\_\_():**

The method **`object.\_\_getstate\_\_()`** is invoked if implemented by classes. When called, it returns an object that is pickled as the content for the instance. This replaces the default behavior of pickling the contents of the instance’s dictionary.

**object.\_\_setstate\_\_(state):**

The method **`object.\_\_setstate\_\_(state)`** is invoked when defined by classes, and it receives the unpickled state as an argument. The pickled state is expected to be a dictionary, and its key-value pairs are assigned to the dictionary of the new instance during unpickling.

**object.\_\_reduce\_\_()**

The `**\_\_reduce\_\_**` method, which accepts no arguments, should return a string or, preferably, a tuple. This method plays a crucial role in defining the pickling process for an object, providing a means to customize the serialization behavior.

**object.\_\_reduce\_ex\_\_(protocol)**

The **`object.\_\_reduce\_ex\_\_(protocol)`** method, akin to **`\_\_reduce\_\_`**, accepts a single integer argument. Its primary purpose is to furnish backward-compatible reduce values tailored for older Python versions. This method is employed to ensure compatibility by offering an alternative representation of the object suitable for earlier Python releases.

**Illustration: Customizing Serialization for Stateful Objects**

In this instance, we showcase the adjustment of pickling behavior for a class. The TextReader class, designed to read a text file while providing line numbers and content through its **`readline()`** method, undergoes a customized serialization and deserialization process using the pickle module. During pickling, only attributes excluding the file object are retained. Upon unpickling, the file is reopened, allowing reading to resume from the previous location.

The provided code defines the TextReader class, exhibits its custom pickling procedures, and demonstrates successful preservation of its state through pickling and unpickling. This ensures seamless continuation of line reading from the last recorded position.

import pickle  
  
class TextReader:  
 def \_\_init\_\_(self, filename):  
 self.filename = filename  
 with open(filename) as file:  
 self.file\_content = file.readlines()  
 self.lineno = 0  
  
 def readline(self):  
 self.lineno += 1  
 if self.lineno <= len(self.file\_content):  
 return f"{self.lineno}: {self.file\_content[self.lineno - 1].rstrip()}"  
 return None  
  
 def \_\_getstate\_\_(self):  
 state = {'filename': self.filename, 'lineno': self.lineno}  
 return state  
  
 def \_\_setstate\_\_(self, state):  
 self.filename = state['filename']  
 self.lineno = state['lineno']  
 with open(self.filename) as file:  
 self.file\_content = file.readlines()  
  
# Testing the TextReader class  
reader = TextReader("example.txt")  
print(reader.readline())  
print(reader.readline())  
  
# Pickle and unpickle the TextReader instance  
new\_reader = pickle.loads(pickle.dumps(reader))  
print(new\_reader.readline())

**JSON Module in Python**

JSON (JavaScript Object Notation) is a popular, lightweight data interchange standard. It represents data structures made up of key-value pairs that are quite straightforward and human-readable. JSON has become the industry standard for data interchange between online services and is widely utilized in modern programming languages, including Python.

**Importing the JSON Module**

In Python, the JSON module is a built-in package that can be used to work with JSON data. To use it, you simply need to import it at the beginning of your script.

import json

**Basic Operations with JSON**

The JSON module in Python provides a powerful set of methods and classes that make working with JSON data simple. Here are some basic operations:

**Encoding Python objects into JSON**

Python objects can be converted into JSON strings using json.dumps() method.

import json  
data = {  
 "name": "John",  
 "age": 30,  
 "city": "New York"  
}  
json\_data = json.dumps(data)  
print(json\_data)

**Output:**  
{“name”: “John”, “age”: 30, “city”: “New York”}

**Decoding JSON into Python objects**

JSON strings can be converted back into Python objects using json.loads() method.

import json  
json\_data = '{"name": "John", "age": 30, "city": "New York"}'  
data = json.loads(json\_data)  
print(data)

**Output:**  
{‘name’: ‘John’, ‘age’: 30, ‘city’: ‘New York’}

**Working with Files**

The JSON module also provides methods for reading and writing JSON data from and to files. The json.dump() function writes JSON data to a file, and json.load() reads JSON data from a file.

import json  
data = {  
 "name": "John",  
 "age": 30,  
 "city": "New York"  
}  
with open('data.txt', 'w') as f:  
 json.dump(data, f)  
  
with open('data.txt', 'r') as f:  
 data = json.load(f)  
print(data)

**Output:**{‘name’: ‘John’, ‘age’: 30, ‘city’: ‘New York’}

**Handling Complex Data Types**

Python’s JSON module supports most of the Python data types. However, it does not support the datetime type. For custom serialization, you can provide a function to the default parameter of json.dumps() or json.dump().

import json  
from datetime import datetime  
  
def datetime\_handler(obj):  
 if isinstance(obj, datetime):  
 return obj.isoformat()  
 raise TypeError("Type %s not serializable" % type(obj))  
  
data = {  
 "name": "John",  
 "age": 30,  
 "birthday": datetime.now()  
}  
  
json\_data = json.dumps(data, default=datetime\_handler)  
print(json\_data)

**Output:**

{“name”: “John”, “age”: 30, “birthday”: “2024–01–03T18:23:38.717554”}

In this example, the datetime\_handler function is used to convert datetime objects into ISO format strings.

**Conclusion**

The JSON module in Python provides a powerful and flexible way to work with JSON data. Whether you need to convert Python objects to JSON strings, or vice versa, the JSON module has got you covered.

**Marshal Module in Python**

The marshal module in Python is used for internal object serialization, meaning it converts Python objects into a binary format that can be saved to disk or transmitted over a network. This module is primarily used by Python itself to support read/write operations on compiled versions of Python modules (.pyc files).

**Importing the Marshal Module**

To use the marshal module, you need to import it at the beginning of your script.

import marshal

**Basic Operations with Marshal**

The marshal module provides several methods for working with serialized data. Here are some basic operations:

**Dumping Python objects into binary format**

Python objects can be converted into binary format using marshal.dumps() method.

import marshal  
data = ["Hello", "World"]  
binary\_data = marshal.dumps(data)  
print(binary\_data)

**Output:**

b’\xdb\x02\x00\x00\x00\xda\x05Hello\xda\x05World’

**Loading binary data into Python objects**

Binary data can be converted back into Python objects using marshal.loads() method.

import marshal  
binary\_data = b'\xdb\x02\x00\x00\x00\xda\x05Hello\xda\x05World'  
data = marshal.loads(binary\_data)  
print(data)

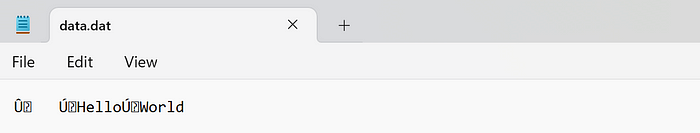
**Output:**

[‘Hello’, ‘World’]

**Writing Binary Data to a File**

The marshal.dump() function writes binary data to a file.

import marshal  
data = ["Hello", "World"]  
with open('data.dat', 'wb') as f:  
 marshal.dump(data, f)



**Reading Binary Data from a File**

The marshal.load() function reads binary data from a file.

import marshal  
with open('data.dat', 'rb') as f:  
 data = marshal.load(f)  
print(data)

**Output:**  
[‘Hello’, ‘World’]

**Example of serializing and deserializing using marshal:**

import marshal   
   
info = { "id": "5",  
 "name": "ABC",  
 "pass": "abc#123"}  
   
# dumping the information. It returns the byte object stored in variable 'marshel\_obj'  
marshel\_obj = marshal.dumps(info)   
print('Serialized Object: ', marshel\_obj)  
   
# loading the byte object into original value  
unmarshal\_obj = marshal.loads(marshel\_obj)   
print('Deserialized Object : ', unmarshal\_obj)

**Output:**

Serialized Object: b'\xfb\xda\x02id\xda\x015\xda\x04name\xda\x03ABC\xda\x04pass\xfa\x07abc#1230'  
Deserialized Object : {'id': '5', 'name': 'ABC', 'pass': 'abc#123'}

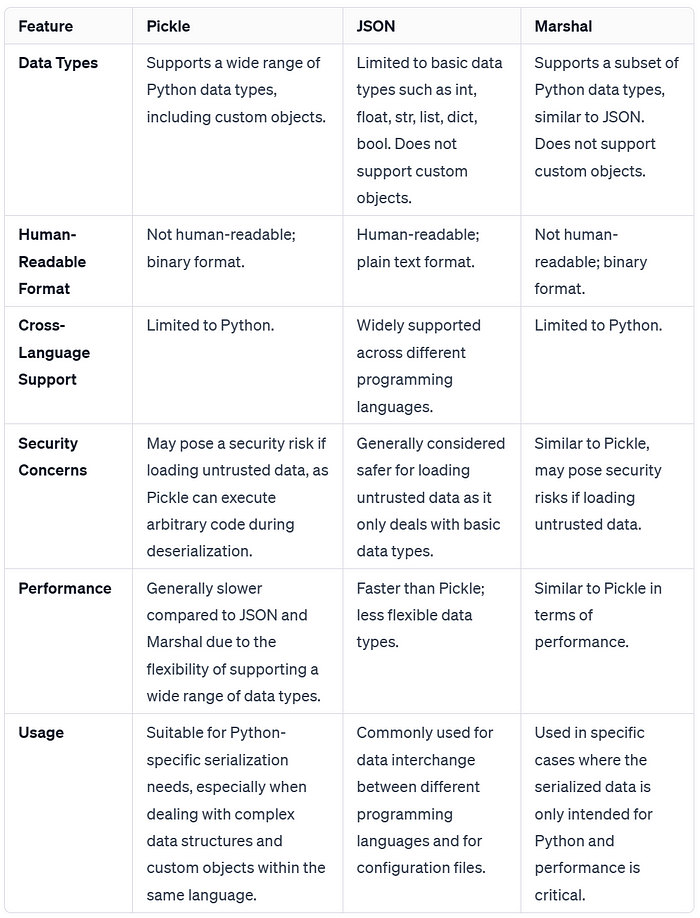
**Limitations of Marshal Module**

The marshal module is not intended to be a general persistence module. It is not secure against erroneous or maliciously constructed data. You should never unmarshal data received from an untrusted or unauthenticated source. Also, not all Python object types are supported; in general, only objects whose value is independent of a particular invocation of Python can be written and read by this module.

**Conclusion**

The marshal module in Python provides a way to serialize and deserialize Python objects into a binary format. However, due to its limitations and lack of security against erroneous or malicious data, it is generally recommended to use other modules like Pickle for such purposes.

**Comparing Pickle with JSON and Marshal in Python**



Serialization is quite handy in case you want to store an object, say you want to use the same object created using token to place orders in various systems, when you are forbidden to create the same object twice.

If you enjoyed this article, give me a follow and check out my profile for more such insightful tech content. Happy learning !!!

[Python](https://medium.com/tag/python?source=post_page-----297a4489ba87---------------python-----------------)

[Serialization](https://medium.com/tag/serialization?source=post_page-----297a4489ba87---------------serialization-----------------)

[Pickles](https://medium.com/tag/pickles?source=post_page-----297a4489ba87---------------pickles-----------------)