**SERIALIZATION**

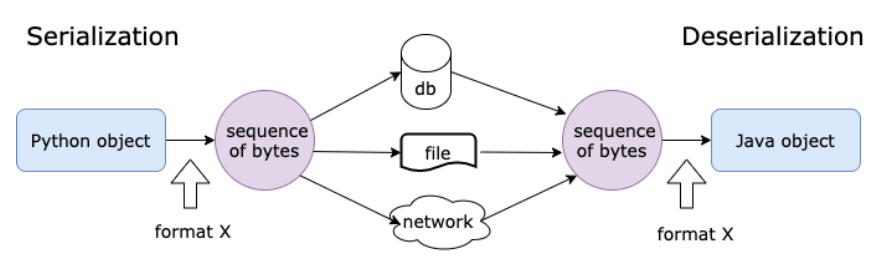
**Why Serialization?**

We create plenty of objects in Python every day, and these objects will eventually disappear if the program dies. Sometimes we want to store the state of an object in a file or in a database, or transmit it across the network for using it in the future. This helps us to reuse the object in different programs or even in different environments.

But the problem we might face is the compatibility. Imagine that you’ve created an object in Python, how could it be reused by a Java program? Boolean values in Python look like True and False, but in Java, they are represented using true and false. There must be a “middle man” creating a universal language that both programs understand — which is a sequence of bytes.

**What is Serialization in Python?**

* In Python, serialization allows you to take a complex object structure and transform it into a stream of bytes that can be saved to a disk or sent over a network. You may also see this process referred to as **marshalling**.
* The reverse process, which takes a stream of bytes and converts it back into a data structure, is called **deserialization** or **unmarshalling**.



There are two groups of serialization format:

* 1. text-based and
  2. binary-based.

Typical text-based serialization formats are CSV, JSON, XML, YAML, TOML, etc. Binary-based formats are protobuf and Avro. Python also has several packages like pickle, numpy and pandas that supports serializing custom objects into byte format.

Python offers three different [modules](https://realpython.com/python-modules-packages/) in the standard library that allow you to serialize and deserialize objects:

1. The [**marshal**](https://docs.python.org/3/library/marshal.html) module
2. The **[json](https://docs.python.org/3/library/json.html)** module
3. The [**pickle**](https://docs.python.org/3/library/pickle.html) module

* The **marshal** module is the oldest of the three listed above. It exists mainly to read and write the compiled bytecode of Python modules, or the .pyc files you get when the interpreter [imports](https://realpython.com/absolute-vs-relative-python-imports/) a Python module. So, even though you can use marshal to serialize some of your objects, it’s not recommended.
* The **json** module is the newest of the three. It allows you to work with standard JSON files. JSON is a very convenient and widely used format for data exchange.

**PICKLE MODULE**

* If we want to serialize and de-serialize a Python object, we use functions and methods from the module **Python Pickle.**
* Pickling, is the act of converting a Python object into a byte stream i.e.  serializes objects in a binary format, which means the result is not human readable.
* We also call this ‘serialization’,  ‘marshalling’, or ‘flattening’.
* Unpickling is its inverse, ie., converting a byte stream from a binary file or bytes-like object into an object.

What Pickle can it do?

* ∙ Pickle can store and reproduce dictionaries and lists very easily.
* ∙ Stores object attributes and restores them back to the same State.

What pickle can’t do?

* ∙ It does not save an objects code. Only it’s attributes values.
* ∙ It cannot store file handles or connection sockets.

Therefore pickling is a way to store and retrieve data variables into and out from files where variables can be lists, classes, etc.

**Pickling steps:**

1. import pickle
2. Write a variable to file, something like

pickle.dump(mystring, outfile, protocol),

where 3rd argument protocol is optional.

**UnPickling steps:**

1. Import pickle
2. Write a variable to a file, something like

myString = pickle.load(inputfile)

**Inside the Python pickle Module**

The Python pickle module basically consists of four methods:

1. pickle.dump(obj, file, protocol=None, \*, fix\_imports=True, buffer\_callback=None)
2. pickle.dumps(obj, protocol=None, \*, fix\_imports=True, buffer\_callback=None)
3. pickle.load(file, \*, fix\_imports=True, encoding="ASCII", errors="strict", buffers=None)
4. pickle.loads(bytes\_object, \*, fix\_imports=True, encoding="ASCII", errors="strict", buffers=None)

The first two methods are used during the pickling process, and the other two are used during unpickling. The only difference between dump() and dumps() is that the first creates a file containing the serialization result, whereas the second returns a string.

To differentiate dumps() from dump(), it’s helpful to remember that the s at the end of the function name stands for string. The same concept also applies to load() and loads(): The first one reads a file to start the unpickling process, and the second one operates on a string.

**Advantages of serialization**

* It is easy to use and can be customized.
* The serialized stream can be encrypted, authenticated and compressed.
* Serialization can also be used as a mechanism for exchanging objects between Java and  C++ libraries.

However, **serialization has some disadvantages too**:

It should ideally not be used with large-sized objects, as it offers significant overhead.  Large objects also significantly increase the memory requirements of your application

Here are three general guidelines for deciding which approach to use:

1. Don’t use the **marshal** module. It’s used mainly by the interpreter, and the official documentation warns that the Python maintainers may modify the format in backward-incompatible ways.
2. The **json** module and XML are good choices if you need interoperability with different languages or a human-readable format.
3. The Python **pickle** module is a better choice for all the remaining use cases. If you don’t need a human-readable format or a standard interoperable format, or if you need to serialize custom objects, then go with pickle.

**Example 1:**



**Example 2:**

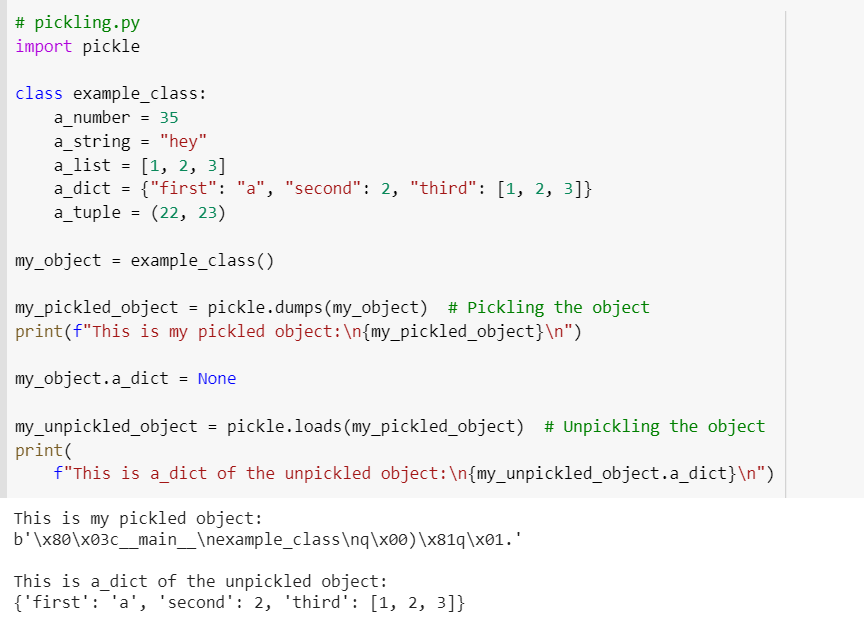


**Example 3:**

Consider the following example. Say you have a custom-defined class named example\_class with several different attributes, each of a different type:

* a\_number
* a\_string
* a\_dictionary
* a\_list
* a\_tuple

The example below shows how you can instantiate the class and pickle the instance to get a plain string. After pickling the class, you can change the value of its attributes without affecting the pickled string. You can then unpickle the pickled string in another [variable](https://realpython.com/python-variables/), restoring an exact copy of the previously pickled class:



The pickling process ends correctly, storing your entire instance in this string: b'\x80\x03c\_\_main\_\_\nexample\_class\nq\x00)\x81q\x01.' After the pickling process ends, you modify your original object by setting the attribute a\_dict to None.

Finally, you unpickle the string to a completely new instance. What you get is a [deep copy](https://realpython.com/copying-python-objects/) of your original object structure from the time that the pickling process began.

**Try:**

1. Unpickle a\_number, a\_string, a\_list, a\_tuple from example 3.