IMPORTING & CLEANING DATA +1

Pickle in Python: Object Serialization

Discover the Python pickle module: learn about serialization, when (not) to use it, how to compress pickled objects, multiprocessing, and much more!

As a data scientist, you will use sets of data in the form of dictionaries, DataFrames, or any other data type. When working with those, you might want to save them to a file, so you can use them later on or send them to someone else. This is what Python's `pickle` module is for: it serializes objects so they can be saved to a file, and loaded in a program again later on.

In this tutorial, you will cover the following topics:

- What is pickling?
- What can pickle be used for?
- When not to use pickle
- What can be pickled?
- Pickle vs JSON
- Pickling files
- Unpickling files
- Compressing pickle files
- Unpickling Python 2 objects in Python 3
- Pickle and multiprocessing

What is pickling?

Pickle is used for serializing and de-serializing Python object structures, also called marshalling or flattening. Serialization refers to the process of converting an object in memory to a byte stream that can be stored on disk or sent over a network. Later on, this character stream can then be retrieved and de-serialized back to a Python object. Pickling is not to be confused with compression! The former is the conversion of an object from one representation (data in Random Access Memory (RAM)) to another (text on disk), while the latter is the process of encoding data with fewer bits, in order to save disk space.

What Can You Do With pickle?

Pickling is useful for applications where you need some degree of persistency in your data. Your program's state data can be saved to disk, so you can continue working on it later on. It can also be used to send data over a Transmission Control Protocol (TCP) or socket connection, or to store python objects in a database. Pickle is very useful for when you're working with machine learning algorithms, where you want to save them to be able to make new predictions at a later time, without having to rewrite everything or train the model all over again.

When Not To Use `pickle`

If you want to use data across different programming languages, pickle is not recommended. Its protocol is specific to Python, thus, cross-language compatibility is not guaranteed. The same holds for different versions of Python itself. Unpickling a file that was pickled in a different version of Python may not always work properly, so you have to make sure that you're using the same version and perform an update if necessary. You should also try not to unpickle data from an untrusted source. Malicious code inside the file might be executed upon unpickling.

Storing data with pickle

What can be pickled?

You can pickle objects with the following data types:

- Integers,
- Floats,
- Complex numbers,
- (normal and Unicode) Strings,
- Tuples,
- Lists,
- Sets, and
- Dictionaries that ontain picklable objects.

All the above can be pickled, but you can also do the same for classes and functions, for example, if they are defined at the top level of a module.

Not everything can be pickled (easily), though: examples of this are generators, inner classes, lambda functions and defaultdicts. In the case of lambda functions, you need to use an additional package named `dill`. With defaultdicts, you need to create them with a module-level function.

Pickle vs JSON

JSON stands for JavaScript Object Notation. It's a lightweight format for data-interchange, that is easily readable by humans. Although it was derived from JavaScript, JSON is standardized and language-independent. This is a serious advantage over pickle. It's also more secure and much faster than pickle.

However, if you only need to use Python, then the `pickle` module is still a good choice for its ease of use and ability to reconstruct complete Python objects.

An alternative is cPickle. It is nearly identical to `pickle`, but written in C, which makes it up to 1000 times faster. For small files, however, you won't notice the difference in speed. Both produce the same data streams, which means that Pickle and cPickle can use the same files.

Pickling files

```
import pickle
```

For this tutorial, you will be pickling a simple dictionary. A dictionary is a list of `key : value` elements. You will save it to a file and then load again. Declare the dictionary as such:

```
dogs_dict = { 'Ozzy': 3, 'Filou': 8, 'Luna': 5, 'Skippy': 10, 'Barco': 12, 'Balou': 9, 'Lai
```

To pickle this dictionary, you first need to specify the name of the file you will write it to, which is `dogs` in this case.

Note that the file does not have an extension.

To open the file for writing, simply use the `open()` function. The first argument should be the name of your file. The second argument is `'wb'`. The `w` means that you'll be writing to the file, and `b` refers to binary mode. This means that the data will be written in the form of byte objects. If you forget the `b`, a `TypeError: must be str, not bytes` will be returned. You may sometimes come across a slightly different notation; `w+b`, but don't worry, it provides the same functionality.

```
filename = 'dogs'
outfile = open(filename,'wb')
```

Once the file is opened for writing, you can use `pickle.dump()`, which takes two arguments: the object you want to pickle and the file to which the object has to be saved. In this case, the former will be `dogs_dict`, while the latter will be `outfile`.

Don't forget to close the file with `close()`!

```
pickle.dump(dogs_dict,outfile)
outfile.close()
```

Unpickling files

The process of loading a pickled file back into a Python program is similar to the one you saw previously: use the `open()` function again, but this time with `'rb'` as second argument (instead of `wb`). The `r` stands for read mode and the `b` stands for binary mode. You'll be reading a binary file. Assign this to `infile`. Next, use `pickle.load()`, with `infile` as argument, and assign it to `new_dict`. The contents of the file are now assigned to this new variable. Again, you'll need to close the file at the end.

```
infile = open(filename,'rb')
new_dict = pickle.load(infile)
infile.close()
```

To make sure that you successfully unpickled it, you can print the dictionary, compare it to the previous dictionary and check its type with `type()`.

```
print(new_dict)
print(new_dict==dogs_dict)
print(type(new_dict))
```

```
{'Ozzy': 3, 'Filou': 8, 'Luna': 5, 'Skippy': 10, 'Barco': 12, 'Balou': 9, 'Laika': 16}
True
<class 'dict'>
```

Compressing pickle files

If you are saving a large dataset and your pickled file takes up a lot of space, you may want to compress it. This can be done using `bzip2` or `gzip`. They both compress files, but `bzip2` is a bit slower. `gzip`, however, produces files about twice as large as `bzip2`. You'll be using `bzip2` in this tutorial.

Remember that compression and serialization is not the same! You can go back to the beginning of the tutorial to refresh your memory if necessary.

```
import bz2
import pickle

sfile = bz2.BZ2File('smallerfile', 'w')
pickle.dump(dogs_dict, sfile)
```

A new file named `smallerfile` should have appeared. Keep in mind that the difference in file size compared to an uncompressed version will not be noticeable with small object structures.

Unpickling Python 2 objects in Python 3

You might sometimes come across objects that were pickled in Python 2 while running Python 3. This can be a hassle to unpickle.

You could either unpickle it by running Python 2, or do it in Python 3 with `encoding='latin1'` in the `load()` function.

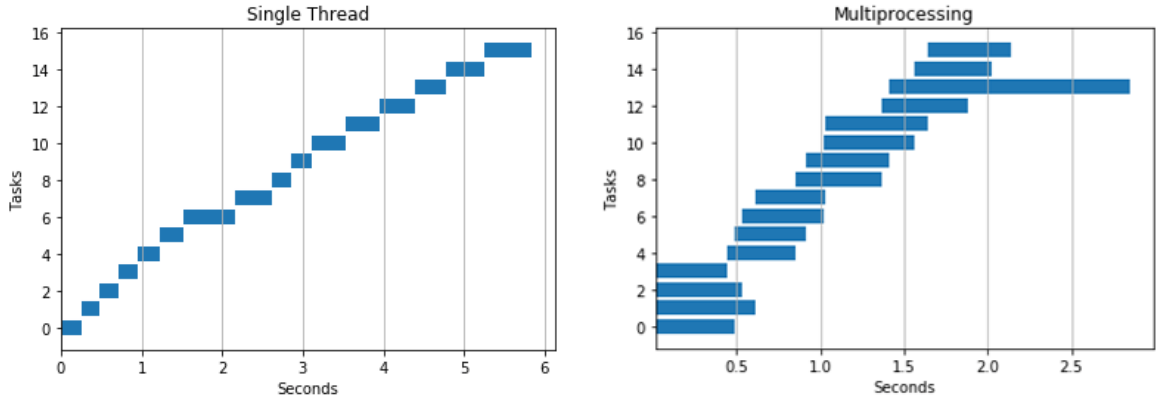
```
infile = open(filename,'rb')
new_dict = pickle.load(infile, encoding='latin1')
```

This will not work if your objects contains NumPy arrays. In that case, you could also try using `encoding='bytes'` :

```
infile = open(filename,'rb')
new_dict = pickle.load(infile, encoding='bytes')
```

Pickle and multiprocessing

When very complex computations have to be executed, it is common to distribute this task over several processes. Multiprocessing means that several processes are executed simultaneously, usually over several Central Processing Units (CPUs) or CPU cores, thus saving time. An example is the training of machine learning models or neural networks, which are intensive and time-consuming processes. By distributing these on a large



Adapted from the graphic presented here.

When a task is divided over several processes, these might need to share data. Processes do not share memory space, so when they have to send information to each other, they use serialization, which is done using the `pickle` module.

In the following example, start my importing `multiprocessing` as `mp`, and `cos` from `math`. Then, create a `Pool` abstraction, where you can specify the amount of processors to use. The `Pool` will handle the multiprocessing in the background. Next, you can map the `cos` function on a range of 10 to the `Pool`, so it can be executed.

```
import multiprocessing as mp
from math import cos

p = mp.Pool(2)

p.map(cos, range(10))
```

```
[1.0,
 0.5403023058681398,
 -0.4161468365471424,
 -0.9899924966004454,
 -0.6536436208636119,
 0.2836621854632263,
 0.9601702866503661,
 0.7539022543433046,
```

As you can see, the `cos` function is perfectly executed. However, this will not always work. Remember that lambda functions can't be pickled. So if you try to apply multiprocessing to a lambda function, it will fail.

```
p.map(lambda x: 2**x, range(10))
```

```
PicklingError: Can't pickle <function <lambda> at 0x111d0a7b8>: attribute lookup <lambda> o
```

There is a solution for this. `dill` is a package similar to `pickle` that can serialize lambda functions, among other things. Its use it almost identical to `pickle` .

```
import dill

dill.dump(lambda x: x**2, open('dillfile','wb'))
```

To use multiprocessing with a lambda function, or other data types unsupported by pickle, you will have to use a fork of `multiprocessing` called `pathos.multiprocessing` . This package uses `dill` for serialization instead of `pickle` . Creating a `Pool` and mapping a lambda function to it is done exactly the same way as you saw before.

```
import pathos.multiprocessing as mp

p = mp.Pool(2)

p.map(lambda x: 2**x, range(10))
```

```
[1, 2, 4, 8, 16, 32, 64, 128, 256, 512]
```

This time, there is no `PicklingError` !

Conclusion

you would like to learn more on how to build such predictive models in Python, you should definitely take a look at our Supervised learning with scikit-learn course. It will teach you all the basics of machine learning in Python.