4.2 Serialization of Python objects using the pickle module

In this section, you will learn how to persist Python objects for later use.

Pickling is the process of preserving or extending the lifespan of food. The resulting food is called a pickle, and to prevent ambiguity, prefaced with the 'pickled' adjective.

Have you ever considered saving the output of your data processing for later use?

The simplest way to persist outcomes is to generate a flat text file and to write your outcomes. It’s a very simple thing to do way which is not suitable for persisting sets of different types of objects or nested structures.

In Python, object **serialization** is the process of converting an object structure into a stream of bytes to store the object in a file or database, or to transmit it via a network. This byte stream contains all the information necessary to reconstruct the object in another Python script.

This reverse process is called **deserialization**.

Python objects can also be serialized using a module called 'pickle', and using this module, you can 'pickle' your Python objects for later use.

The 'pickle' module is a very popular and convinient module for data serialization in the world of Pythonistas.

So, what can be pickled and then unpickled?

The following types can be pickled:

- None, booleans;
- integers, floating-point numbers, complex numbers;
- strings, bytes, bytearrays;
- tuples, lists, sets, and dictionaries containing pickleable objects;
- objects, including objects with references to other objects (remember to avoid cycles!)
- references to functions and classes, but not their definitions.

Let's pickle our first set of data consisting of:

- a nested dictionary carrying some information about currencies;
- a list containing a string, an integer, and a list.

When you run the code presented in the right pane, a new file should be created. Remember to run the code locally.

```
import pickle

a_dict = dict()
a_dict['EUR'] = {'code':'Euro', 'symbol': '€'}
a_dict['GBP'] = {'code':'Pounds sterling', 'symbol': '£'}
a_dict['USD'] = {'code':'US dollar', 'symbol': '$'}
a_dict['JPY'] = {'code':'Japanese yen', 'symbol': '¥'}

a_list = ['a', 123, [10, 100, 1000]]

with open('multidata.pckl', 'wb') as file_out:
    pickle.dump(a_dict, file_out)
```

```
    pickle.dump(a_list, file_out)
```

The code starts with the import statement responsible for loading the pickle module:

```
import pickle
```

Later you can see that the file handle 'file_out' is associated with the file opened for writing in binary mode. It's important to open the file in binary mode as we are dumping data as a stream of bytes.

```
with open('multidata.pckl', 'wb') as file_out:
```

Now it's time to persist the first object with the `dump()` function. This function expects an object to be persisted and a file handle.

```
pickle.dump(a_dict, file_out)
```

And the second object is persisted in the same way:

```
pickle.dump(a_list, file_out)
```

Now it's time to unpickle the contents of the file.

```
import pickle

with open('multidata.pckl', 'rb') as file_in:
    data1 = pickle.load(file_in)
    data2 = pickle.load(file_in)

print(type(data1))
print(data1)
print(type(data2))
print(data2)
```

The presented code is quite simple:

- we're importing a pickle module;
- the file is opened in binary mode and the file handle is associated with the file;
- we consecutively read some portions of data and deserialize it with the `load()` function;
- finally, we examine the type and contents of the objects.

Pay attention to the fact that with the 'pickle' module, you have to remember the order in which the objects were persisted and the deserialization code should follow the same order.

output

```
<class 'dict'>
{'EUR': {'code': 'Euro', 'symbol': '€'}, 'GBP': {'code': 'Pounds sterling',
'symbol': '£'}, 'USD': {'code': 'US dollar', 'symbol': '$'}, 'JPY': {'code':
'Japanese yen', 'symbol': '¥'}}
<class 'list'>
['a', 123, [10, 100, 1000]]
```

At the beginning of the serialization module, we mentioned that serialized objects could be persisted in a database or sent via a network. This implies another two functions corresponding to the `pickle.dumps()` and

`pickle.loads()` functions:

- `pickle.dumps(object_to_be_pickled)` – expects an initial object, returns a byte object. This byte object should be passed to a database or network driver to persist the data;
- `pickle.loads(bytes_object)` – expects the bytes object, returns the initial object.

An example of in situ serialization and deserialization is presented in the right pane.

```
import pickle

a_list = ['a', 123, [10, 100, 1000]]
bytes = pickle.dumps(a_list)
print('Intermediate object type, used to preserve data:', type(bytes))

# now pass 'bytes' to appropriate driver

# therefore when you receive a bytes object from an appropriate driver you can
deserialize it
b_list = pickle.loads(bytes)
print('A type of deserialized object:', type(b_list))
print('Contents:', b_list)
```

output

```
Intermediate object type, used to preserve data: <class 'bytes'>
A type of deserialized object: <class 'list'>
Contents: ['a', 123, [10, 100, 1000]]
```

Remember that attempts to pickle non-pickleable objects will raise the `PicklingError` exception.

Trying to pickle a highly recursive data structure (mind the cycles) may exceed the maximum recursion depth, and a `RecursionError` exception will be raised in such cases.

Note that functions (both built-in and user-defined) are pickled by their name reference, not by any value. This means that only the function name is pickled; neither the function's code, nor any of its function attributes, are pickled.

Similarly, classes are pickled by named reference, so the same restrictions in the unpickling environment apply. Note that none of the class's code or data are pickled.

This is done on purpose, so you can fix bugs in a class or add methods to the class, and still load objects that were created with an earlier version of the class.

Hence, your role is to ensure that the environment where the class or function is unpickled is able to import the class or function definition. In other words, the function or class must be available in the namespace of your code reading the pickle file.

Otherwise, an `AtrributeError` exception will be raised.

The following code demonstrates the situation for function definition pickling:

```
import pickle

def f1():
    print('Hello from the jar!')
```

```
with open('function.pckl', 'wb') as file_out:
    pickle.dump(f1, file_out)
```

We see no errors, so we might conclude that `f1()` was pickled successfully, and now we can retrieve it from the file.

Run the code in the editor and see what happens.

```
import pickle

with open('function.pckl', 'rb') as file_in:
    data = pickle.load(file_in)

print(type(data))
print(data)
data()
```

Unfortunately, the result proves that no code was persisted:

output

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    data = pickle.load(file_in)
AttributeError: Can't get attribute 'f1' on <module '__main__' from 'main.py'>
```

Here's the same example regarding class definition and object pickling:

```
import pickle

class Cucumber:
    def __init__(self):
        self.size = 'small'

    def get_size(self):
        return self.size

cucu = Cucumber()

with open('cucumber.pckl', 'wb') as file_out:
    pickle.dump(cucu, file_out)
```

We see no errors, so we might conclude that the `Cucumber` class and object were pickled successfully, and now we can retrieve them from the file. In fact, only the object is persisted but not its definition allowing us to determine the attribute layout:

```
import pickle

with open('cucumber.pckl', 'rb') as file_in:
    data = pickle.load(file_in)

print(type(data))
print(data)
print(data.size)
```

```
print(data.get_size())
```

If you run the code, you receive:

output

```
Traceback (most recent call last):
  File "main.py", line 4, in <module>
    data = pickle.load(file_in)
AttributeError: Can't get attribute 'Cucumber' on <module '__main__' from
'main.py'>
```

The remedy for the above problems is: the code that calls the `load()` or `loads()` functions of `pickle` should already know the function/class definition.

A few additional words about the pickle module:

- it's a Python implementation of the serialization process, so the pickled data cannot be exchanged with programs written in other languages like Java or C++. In such cases, you should think about the JSON or XML formats, which could be less convenient than pickling, but when assimilated are more powerful than pickling;
- the pickle module is constantly evolving, so the binary format may differ between different versions of Python. Pay attention that both serializing and deserializing parties are making use of the same pickle versions;
- the pickle module is not secured against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.