Chapter 11

Serialization

The term *serialization* refers to the process of transforming any object into a sequence of bytes to be able to storage or transfer its data. We often use serialization to keep the results or states after a program finishes its execution. It may be very useful when another program or a later execution of the same program can load the saved objects and reuse them.

The Python `pickle` module allows us to serialize and deserialize objects. This module provides two principal methods

1. `dumps()` method: allows us to serialize an object.
2. `loads()` method: let us to deserialize the data and return the *original* object.

```
1 # 29.py
2
3 import pickle
4
5 tuple_ = ("a", 1, 3, "hi")
6 serial = pickle.dumps(tuple_)
7 print(serial)
8 print(type(serial))
9 print(pickle.loads(serial))
```

```
b'\x80\x03(X\x01\x00\x00\x00aq\x00K\x01K\x03X\x02\x00\x00\x00hiq\x01tq\x02.'
<class 'bytes'>
('a', 1, 3, 'hi')
```

Pickle has also the `dump()` and `load()` methods to serialize and deserialize through **files**. These methods are not the same methods `dumps()` and `loads()` described previously. The `dump()` method saves a file with the serialized object and the `load()` deserializes the content of the file. The following example shows how to use them:

```
# 30.py

import pickle

list_ = [1, 2, 3, 7, 8, 3]
with open("my_list", 'wb') as file:
    pickle.dump(list_, file)

with open("my_list", 'rb') as file:
    my_list = pickle.load(file)
    # This will generate an error if the object is not same we saved
    assert my_list == list_
```

The pickle module **is not safe**. You should never load a pickle file when you do not know its origin since it could run malicious code on your computer. We will not go into details on how to inject code via the pickle module, we refer the reader to [2] for more information about this topic. If we use Python 3 to serialize an object that will be deserialized later in Python 2, we have to pass an extra argument to `dump` or `dumps` functions, the argument name is `protocol` and must be equal to `2`. The default value is `3`). The next example shows how to change the pickle protocol:

```
# 31.py

import pickle

my_object = [1, 2, 3, 4]
serial = pickle.dumps(my_object, protocol=2)
```

When pickle is serializing an object, what is trying to do is to save the attribute `__dict__` of the object. Interestingly, before checking the attribute `__dict__`, pickle checks if there is a method called `__getstate__`, if any, it will serialize what the method `__getstate__` returns instead of the dictionary `__dict__` of the object. It allows us to customize the serialization:

```
# 32.py
```

```
2
3  import pickle
4
5
6  class Person:
7
8      def __init__(self, name, age):
9          self.name = name
10         self.age = age
11         self.message = "Nothing happens"
12
13     # Returns the current object state to be serialized by pickle
14     def __getstate__(self):
15         # Here we create a copy of the current dictionary, to modify the copy,
16         # not the original object
17         new = self.__dict__.copy()
18         new.update({"message": "I'm being serialized!!"})
19         return new
20
21 m = Person("Bob", 30)
22 print(m.message)
23 serial = pickle.dumps(m)
24 m2 = pickle.loads(serial)
25 print(m2.message)
26 print(m.message)  # The original object is "the same"
```

```
Nothing happens
I'm being serialized!!
Nothing happens
```

Naturally, we can also customize the serialization by implementing the `__setstate__` method, it will run each time you call `load` or `loads`, for setting the current state of the newly deserialized object. The `__setstate__` method receives as argument the state of the object that was serialized, which corresponds to the value returned by `__getstate__`. `__setstate__` must set the state in which we want the deserialized object to be by setting `self.__dict__`. For instance:

```
1  # 33.py
```

```
2
3  import pickle
4
5
6  class Person:
7
8      def __init__(self, name, age):
9          self.name = name
10         self.age = age
11         self.message = "Nothing happens"
12
13     # Returns the current object state to be serialized by pickle
14     def __getstate__(self):
15         # Here we create a copy of the current dictionary, to modify the copy,
16         #  not the original object
17         new = self.__dict__.copy()
18         new.update({"message": "I'm being serialized!!"})
19         return new
20
21     def __setstate__(self, state):
22         print("deserialized object, setting its state...\n")
23         state.update({"name": state["name"] + " deserialized"})
24         self.__dict__ = state
25
26 m = Person("Bob", 30)
27 print(m.name)
28 serial = pickle.dumps(m)
29 m2 = pickle.loads(serial)
30 print(m2.name)
```

```
Bob
deserialized object, setting its state...

Bob deseialized
```

A practical application of `__getstate__` and `__setstate__` methods can be when we need to serialize an

object that contains attributes that will lose sense after serialization, such as, a database connection. A possible solution is: first to use `__getstate_` to remove the database connection within the serialized object; and then manually reconnect the object during its deserialization, in the `__setstate__` method.

11.1 Serializing web objects with JSON

One disadvantage of pickle serialized objects is that only other Python programs can deserialize them. JavaScript Object Notation (**JSON**) is a standard data exchange format that can be interpreted by many different systems. JSON may also be easily read and understood by humans. The format in which information is stored is very similar to Python dictionaries. JSON can only serialize data (`int`, `str`, `floats`, `dictionaries` and `lists`), therefore, you can not serialize functions or classes. In Python there is a module that transforms data from Python to JSON format, called `json`, which provides an interface similar to `dump(s)` and `load(s)` in pickle. The output of a serialization using the `json` module's `dump` method is of course an object in JSON format. The following code shows an example:

```
1  # 34.py
2
3  import json
4
5
6  class Person:
7
8      def __init__(self, name, age, marital_status):
9          self.name = name
10         self.age = age
11         self.marital_status = marital_status
12         self.idn = next(Person.gen)
13
14     def get_id():
15         cont = 1
16         while True:
17             yield cont
18             cont += 1
19
20     gen = get_id()
21
22 p = Person("Bob", 35, "Single")
```

```
23 json_string = json.dumps(p.__dict__)
24 print("JSON data: ")
25 print(json_string)
26 print("Python data: ")
27 print(json.loads(json_string))
```

```
JSON data:
{"marital_status": "Single", "name": "Bob", "idn": 1, "age": 35}
Python data:
{'marital_status': 'Single', 'name': 'Bob', 'age': 35, 'idn': 1}
```

We can also write directly JSON objects as Python strings that follow the JSON data format. In the next instance we create a JSON object type directly (without `json.dumps`), and then we deserialize it into a Python type object with `json.loads`:

```
1 # 35.py
2
3 import json
4
5
6 json_string = '{"name":"Mark","age":34,' \
7               '"marital_status": "married", "score" : 90.5}'
8 print(json.loads(json_string))
```

```
{'marital_status': 'married', 'age': 34, 'score': 90.5, 'name': 'Mark'}
```

We can also load data with particular formats. For instance, in the case we want to show `int` types as `floats`:

```
1 # 36.py
2
3 import json
4
5
6 json_string = '{"name":"Mark","age":34,' \
7               '"marital_status": "married", "score" : 90.5}'
8 print(json.loads(json_string, parse_int=float))
```

```
{'age': 34.0, 'name': 'Mark', 'score': 90.5, 'marital_status': 'married'}
```

In Python, by default, JSON converts all data to a dictionary. If you want to turn data into another type, we can use the object argument `object_hook` with a `lambda` function that will be applied to each data object. For instance, if we want to load JSON data into a list of tuples instead of a dictionary:

```
# 37.py

import json


json_string = '{"name":"Mark","age":34,' \
              '"marital_status": "married", "score" : 90.5}'
data = json.loads(json_string,
                  object_hook=lambda dict_obj:
                  [tuple((i, j)) for i, j in dict_obj.items()])
print(data)
```

```
[('marital_status', 'married'), ('age', 34), ('name', 'Mark'), ('score', 90.5)]
```

We can create any function and then apply it to the data we want to convert:

```
# 38.py

import json


def funcion(dict_obj):
    collection = []
    for k in dict_obj:
        collection.extend([k, str(dict_obj[k])])
    return collection

json_string = '{"name":"Mark","age":34,' \
              '"marital_status": "married", "score" : 90.5}'
data = json.loads(json_string, object_hook=lambda dict_obj: funcion(dict_obj))
print(data)
```

```
['age', '34', 'name', 'Mark', 'score', '90.5', 'marital_status', 'married']
```

We can also customize the way we code the data in JSON format by creating a class that inherits from the `json.JSONEncoder` class and by overriding the default method:

```
# 39.py

import json
from datetime import datetime


class Person:
    def __init__(self, name, age, marital_status):
        self.name = name
        self.age = age
        self.marital_status = marital_status
        self.idn = next(Person.gen)

    def get_id():
        cont = 1
        while True:
            yield cont
            cont += 1

    gen = get_id()


class PersonaEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, Person):
            return {'Person_id': obj.idn, 'name': obj.name,
                    'age': obj.age, 'marital_status': obj.marital_status,
                    'dob': datetime.now().year - obj.age}
        return super().default(obj)


p1 = Person("Bob", 37, "Single")
p2 = Person("Mark", 33, "Married")
```

```
34 p3 = Person("Peter", 24, "Single")
35
36 print("Default serialization:\n")
37 # With this we serialized using the default method
38 json_string = json.dumps(p1.__dict__)
39 print(json_string)
40
41 #  Now we serialize with the personalized method
42 print("\nCustom Serialization:\n")
43 json_string = json.dumps(p1, cls=PersonaEncoder)
44 print(json_string)
45 json_string = json.dumps(p2, cls=PersonaEncoder)
46 print(json_string)
47 json_string = json.dumps(p3, cls=PersonaEncoder)
48 print(json_string)
```

```
Default serialization:

{"name": "Bob", "marital_status": "Single", "age": 37, "idn": 1}

Custom Serialization:

{"age": 37, "name": "Bob", "marital_status": "Single", "Person_id": 1,
"dob": 1978}
{"age": 33, "name": "Mark", "marital_status": "Married", "Person_id": 2,
"dob": 1982}
{"age": 24, "name": "Peter", "marital_status": "Single", "Person_id": 3,
"dob": 1991}
```

11.2 Hands-On Activities

Activity 11.1

The *Walkcart* supermarket has asked us to help them to handle its clients' transactional data. This information, in future, will be useful to know who are the best customers and cashiers. The *Walkcart* managers asked us to implement

all the needed functionalities to allow the clerk to update and save clients' data. This information has to be stored in a file inside the *ClientsDB* folder. All files need to have the `.walkcart` extension.

Each cashier should be able to:

- Star a new session with their name. To verify that the person is really a cashier, you can verify the file in `cashiers.walkcart` that contains a list of serialized strings.
- For each client, ask its name `(str)`, **id** `(int)` and money spent `(int)`.
- Update and generate the file `id.walkcart` inside the `ClientsDB` folder, where `id` is the client's id. This file must be a serialization of the `Client` class, that you must define. You have to save: *name*, *id*, *accumulated spent*, and **last purchase's date**.

The company should also be able to:

- Star a session with a unique user: *WalkcartUnlimited*.
- Generate the file `TOP.walkcart`. This file must be in `format` and must contain the data of the client who historically has spent the most. If you find more than one winner, pick one randomly.

Notes

- If the username does not correspond to a cashier (or a *WalkcartUnlimited*), then this user must not be able to log into the system.
- At all times you must serialize using `pickle`.
- Each cashier can attend a customer more than once, and one client can make as many purchases as he/she want. Keep the information in the file correct and updated.
- The `PlainTextInfo.txt` file cannot be used by your program.