# Object Serialisation

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

## Introduction

Objects are great, but sometimes we need to move them around. Maybe we want to save an object to a file so we can read it in during a later session. Perhaps we want to send the object over a network to a remote process.

To do this, we need to convert the object into a *serial* format, like a sequence of bytes or a string. We also need a way to convert the object from its serial format into a functioning object instance.

## Approaches to Serialisation

We will look at two ways to serialise Python objects.

- Binary format: Python's `pickle` module
- String format: Python's `json` module and related tooling

## Pickle

Python's standard `pickle` module saves Python built-in types and some custom objects in a binary format.

- Pros:
  - Easy to use
  - Efficient
- Cons:
  - Python-specific
  - Can't save or load some values

Reference: `https://docs.python.org/3/library/pickle.html`

## Pickle Versions

There are currently six versions of the `pickle` protocol

- Versions 0 -2: Too old, don't care
- Version 3: introduced in Python 3.0
- Version 4: introduced in Python 3.4
- Version 5: introduced in Python 3.9

New versions have improved features, but data pickled with a high version can't be unpickled in older interpreters that don't support the protocol version. You can specify the protocol version used when pickling.

## Basic Pickle Functions

- `pickle.dump()`: Saves data to a file
- `pickle.dumps()`: Returns a byte string of pickled data
- `pickle.load()`: Loads from a file
- `pickle.loads()`: Loads from a byte string

## Example

```
import pickle

class PickleMe:
    def __init__(self, data):
        self.data = data

example = PickleMe(42)
pickle.dump(example, open('save.p', 'wb'))

example = None
example = pickle.load(open('save.p', 'rb'))
print(example.data) # prints 42
```

## Security Concerns

Pickled data is vulnerable to be altered. Don't load untrusted pickled data. If you need to send and receive pickled data over an untrusted network use something like the `hmac` module to cryptographically authenticate the data.

## Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `17-practical` in your practicals repo.
3. Add a subdirectory, `17-practical` and copy `17-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook.
5. We will discuss results in 20ish minutes.

## Serialising with JSON

Javascript Object Notation (JSON) is a common data serialisation format. Python's `json` module has functions corresponding to those we saw in `pickle`.

- `json.dump()`: Saves data to a file
- `json.dumps()`: Returns a JSON-formatted string of data
- `json.load()`: Loads from a file
- `json.loads()`: Loads from a JSON string

Reference: `https://docs.python.org/3/library/json.html`

## Pros and Cons

- Pros:
  - Not Python-dependent
  - Easy
  - Extensible
- Cons:
  - Can’t represent some Python values without additional code
  - Can’t represent custom classes without additional code

## Values that serialise to JSON

- integers
- floats
- strings
- booleans
- list
- dictionaries[1]
- `None`

[1]Sort of. JSON dictionaries have to have string keys.

## Example

```
import json

pets = {'cat': 'Lola',
        'dog': 'Star',
        'fish': 'My fish don't have names.'
       }

json_pets = json.dumps(pets)

restored_pets = json.loads(json_pets)
```

## Extending JSONEncoder

If we want to handle a type not supported by default, we can extend the default encoder.

```
import json

z = 1 + 2j  # complex number
class ComplexEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, complex):
            return [obj.real, obj.imag]
        return json.JSONEncoder.default(self, obj)

json_z = json.dumps(z, cls=ComplexEncoder)
```

## Adding Serialisation to a Class

Since the ‘json‘ module can’t handle custom classes, a typical approach is to add serialisation methods to the class.

```
class Foo:
    def __init__(self, a, b):
        self.a = a
        self.b = b

    def dumps(self):
        return json.dumps({'a': self.a, 'b': self.b})

    @classmethod
    def loads(cls, js):
        data = json.loads(js)
        return cls(data['a'], data['b'])
```

We basically just manually make our objects handle their own serialisation.

## More Robust Serialisation

The pip-installable module `marshmallow` provides a more robust approach to JSON (and other) serialisation.

```
from marshmallow import Schema, fields

class Foo:
    def __init__(self, a, b):
        self.a = a
        self.b = b

class FooSchema(Schema):
    a = fields.Str(required=True)
    b = fields.Integer()

    @post_load
    def make_foo(self, data, **kwargs):
        return Foo(**data)
```

## More Robust Serialisation

```
foo = Foo('hello', 6)
foo_schema = FooSchema()
serialised = foo_schema.dump(foo)  # returns a dict, not json
deserialised = foo_schema.load(serialised)
```

Reference:
https://marshmallow.readthedocs.io/en/stable/index.html