Doug Hellmann

# The Python 3 Standard Library by Example

**Musée d’Orsay, Paris, France**
Located on the Seine’s left bank, the Musée d’Orsay is housed in a breathtaking Beaux-Arts building originally designed as the world’s first electrified urban railway station. The original “Gare d’Orsay” was built on the site of the old Palais d’Orsay, which had lain empty since it burned in 1871 during the Paris Commune. The building opened on Bastille Day, July 14, 1900, to help celebrate Paris’s Fifth Universal Exhibition. Designated a Historical Monument in 1978, it was then recreated as a museum by Renaud Bardon, Pierre Colboc, and Jean-Paul Philippon of the ACT architecture group. Per the museum’s official history, the new architects “highlighted the great hall, using it as the main artery of the visit, and transformed the magnificent glass awning into the museum’s entrance.” Inside, Gae Aulenti adapted the enormous station into museum spaces, unified via consistent stone wall and floor surfaces. Opened in 1986, the new museum brought together three major art collections from the era 1848-1914. More than three million visitors now come every year to see works from artists including Cézanne, Courbet, Degas, Gauguin, Manet, Monet, and Renoir.

```
        self.name_backwards = name[::-1]
        return


data = []
data.append(SimpleObject('pickle'))
data.append(SimpleObject('preserve'))
data.append(SimpleObject('last'))

# Simulate a file.
out_s = io.BytesIO()

# Write to the stream.
for o in data:
    print('WRITING : {} ({})'.format(o.name, o.name_backwards))
    pickle.dump(o, out_s)
    out_s.flush()

# Set up a readable stream.
in_s = io.BytesIO(out_s.getvalue())

# Read the data.
while True:
    try:
        o = pickle.load(in_s)
    except EOFError:
        break
    else:
        print('READ    : {} ({})'.format(
            o.name, o.name_backwards))
```

This example simulates streams using two `BytesIO` buffers. The first buffer receives the pickled objects, and its value is fed to a second buffer from which `load()` reads. A simple database format could use pickles to store objects, too. The `shelve` (page 405) module is one such implementation.

```
$ python3 pickle_stream.py

WRITING : pickle (elkcip)
WRITING : preserve (evreserp)
WRITING : last (tsal)
READ    : pickle (elkcip)
READ    : preserve (evreserp)
READ    : last (tsal)
```

Besides storing data, pickles are handy for interprocess communication. For example, `os.fork()` and `os.pipe()` can be used to establish worker processes that read job instructions from one pipe and write the results to another pipe. The core code for managing the worker pool and sending jobs in and receiving responses can be reused, since the job and response objects do not have to be based on a particular class. When using pipes or sockets, do not forget to flush after dumping each object, so as to push the data through the connection to the other end. See the `multiprocessing` (page 586) module for a reusable worker pool manager.

### 7.1.3 Problems Reconstructing Objects

When working with custom classes, the class being pickled must appear in the namespace of the process reading the pickle. Only the data for the instance is pickled, not the class definition. The class name is used to find the constructor to create the new object when unpickling. The following example writes instances of a class to a file.

**Listing 7.4: `pickle_dump_to_file_1.py`**

```
import pickle
import sys


class SimpleObject:

    def __init__(self, name):
        self.name = name
        l = list(name)
        l.reverse()
        self.name_backwards = ''.join(l)


if __name__ == '__main__':
    data = []
    data.append(SimpleObject('pickle'))
    data.append(SimpleObject('preserve'))
    data.append(SimpleObject('last'))

    filename = sys.argv[1]

    with open(filename, 'wb') as out_s:
        for o in data:
            print('WRITING: {} ({})'.format(
                o.name, o.name_backwards))
            pickle.dump(o, out_s)
```

When run, the script creates a file based on the name given as an argument on the command line.

```
$ python3 pickle_dump_to_file_1.py test.dat

WRITING: pickle (elkcip)
WRITING: preserve (evreserp)
WRITING: last (tsal)
```

A simplistic attempt to load the resulting pickled objects fails.

**Listing 7.5: `pickle_load_from_file_1.py`**

```
import pickle
import pprint
import sys

filename = sys.argv[1]

with open(filename, 'rb') as in_s:
    while True:
        try:
            o = pickle.load(in_s)
        except EOFError:
            break
        else:
            print('READ: {} ({})'.format(
                o.name, o.name_backwards))
```

This version fails because there is no `SimpleObject` class available.

```
$ python3 pickle_load_from_file_1.py test.dat

Traceback (most recent call last):
  File "pickle_load_from_file_1.py", line 15, in <module>
    o = pickle.load(in_s)
AttributeError: Can't get attribute 'SimpleObject' on <module '_
_main__' from 'pickle_load_from_file_1.py'>
```

The corrected version, which imports `SimpleObject` from the original script, succeeds. Adding this import statement to the end of the import list allows the script to find the class and construct the object.

```
from pickle_dump_to_file_1 import SimpleObject
```

Running the modified script now produces the desired results.

```
$ python3 pickle_load_from_file_2.py test.dat

READ: pickle (elkcip)
READ: preserve (evreserp)
READ: last (tsal)
```

## 7.1.4 Unpicklable Objects

Not all objects can be pickled. Sockets, file handles, database connections, and other objects with runtime state that depends on the operating system or another process may not be able to be saved in a meaningful way. Objects that have non-picklable attributes can define `__getstate__()` and `__setstate__()` to return a subset of the state of the instance to be pickled.

The `__getstate__()` method must return an object containing the internal state of the object. One convenient way to represent that state is with a dictionary, but the value can be any picklable object. The state is stored, and then passed to `__setstate__()` when the object is loaded from the pickle.

**Listing 7.6: pickle_state.py**

```
import pickle


class State:

    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return 'State({!r})'.format(self.__dict__)


class MyClass:

    def __init__(self, name):
        print('MyClass.__init__({})'.format(name))
        self._set_name(name)

    def _set_name(self, name):
        self.name = name
        self.computed = name[::-1]

    def __repr__(self):
        return 'MyClass({!r}) (computed={!r})'.format(
            self.name, self.computed)

    def __getstate__(self):
        state = State(self.name)
        print('__getstate__ -> {!r}'.format(state))
        return state

    def __setstate__(self, state):
        print('__setstate__({!r})'.format(state))
        self._set_name(state.name)


inst = MyClass('name here')
print('Before:', inst)
```

```
dumped = pickle.dumps(inst)

reloaded = pickle.loads(dumped)
print('After:', reloaded)
```

This example uses a separate `State` object to hold the internal state of `MyClass`. When an instance of `MyClass` is loaded from a pickle, `__setstate__()` is passed a `State` instance that it uses to initialize the object.

```
$ python3 pickle_state.py

MyClass.__init__(name here)
Before: MyClass('name here') (computed='ereh eman')
__getstate__ -> State({'name': 'name here'})
__setstate__(State({'name': 'name here'}))
After: MyClass('name here') (computed='ereh eman')
```

**WARNING**

If the return value is false, then `__setstate__()` is not called when the object is unpickled.

## 7.1.5 Circular References

The pickle protocol automatically handles circular references between objects, so complex data structures do not need any special handling. Consider the directed graph in Figure 7.1. It includes several cycles, yet the correct structure can be pickled and then reloaded.

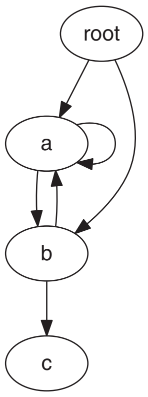

**Figure 7.1: Pickling a Data Structure with Cycles**

**Listing 7.7: pickle_cycle.py**

```
import pickle


class Node:
    """A simple digraph
    """
    def __init__(self, name):
        self.name = name
        self.connections = []

    def add_edge(self, node):
        "Create an edge between this node and the other."
        self.connections.append(node)

    def __iter__(self):
        return iter(self.connections)


def preorder_traversal(root, seen=None, parent=None):
    """Generator function to yield the edges in a graph.
    """
    if seen is None:
        seen = set()
    yield (parent, root)
    if root in seen:
        return
    seen.add(root)
    for node in root:
        recurse = preorder_traversal(node, seen, root)
        for parent, subnode in recurse:
            yield (parent, subnode)


def show_edges(root):
    "Print all the edges in the graph."
    for parent, child in preorder_traversal(root):
        if not parent:
            continue
        print('{:>5} -> {:>2} ({})'.format(
            parent.name, child.name, id(child)))


# Set up the nodes.
root = Node('root')
a = Node('a')
b = Node('b')
c = Node('c')
```