11.1. pickle — Python object serialization

The pickle module implements a fundamental, but powerful algorithm for serializing and de-serializing a Python object structure. “Pickling” is the process whereby a Python object hierarchy is converted into a byte stream, and “unpickling” is the inverse operation, whereby a byte stream is converted back into an object hierarchy. Pickling (and unpickling) is alternatively known as “serialization”, “marshalling,” [1] or “flattening”, however, to avoid confusion, the terms used here are “pickling” and “unpickling”.

This documentation describes both the pickle module and the cPickle module.

Warning: The pickle module is not secure against erroneous or maliciously constructed data. Never unpickle data received from an untrusted or unauthenticated source.

11.1.1. Relationship to other Python modules

The pickle module has an optimized cousin called the cPickle module. As its name implies, cPickle is written in C, so it can be up to 1000 times faster than pickle. However it does not support subclassing of the Pickler() and Unpickler() classes, because in cPickle these are functions, not classes. Most applications have no need for this functionality, and can benefit from the improved performance of cPickle. Other than that, the interfaces of the two modules are nearly identical; the common interface is described in this manual and differences are pointed out where necessary. In the following discussions, we use the term “pickle” to collectively describe the pickle and cPickle modules.

The data streams the two modules produce are guaranteed to be interchangeable.

Python has a more primitive serialization module called marshal, but in general pickle should always be the preferred way to serialize Python objects. marshal exists primarily to support Python’s .pyc files.

The pickle module differs from marshal in several significant ways:

- The pickle module keeps track of the objects it has already serialized, so that later references to the same object won’t be serialized again. marshal doesn’t do this.

  This has implications both for recursive objects and object sharing. Recursive objects are objects that contain references to themselves. These are not handled by marshal, and in fact, attempting to marshal recursive objects will crash your Python interpreter. Object sharing happens when there are multiple references to the same object in different places in the object hierarchy being serialized. pickle stores such objects only once, and ensures that all other references point to the master copy. Shared objects remain shared, which can be very important for mutable objects.

- `marshal` cannot be used to serialize user-defined classes and their instances. `pickle` can save and restore class instances transparently, however the class definition must be importable and live in the same module as when the object was stored.

- The `marshal` serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support `.pyc` files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The `pickle` serialization format is guaranteed to be backwards compatible across Python releases.

Note that serialization is a more primitive notion than persistence; although `pickle` reads and writes file objects, it does not handle the issue of naming persistent objects, nor the (even more complicated) issue of concurrent access to persistent objects. The `pickle` module can transform a complex object into a byte stream and it can transform the byte stream into an object with the same internal structure. Perhaps the most obvious thing to do with these byte streams is to write them onto a file, but it is also conceivable to send them across a network or store them in a database. The module `shelve` provides a simple interface to pickle and unpickle objects on DBM-style database files.

11.1.2. Data stream format

The data format used by `pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as XDR (which can't represent pointer sharing); however it means that non-Python programs may not be able to reconstruct pickled Python objects.

By default, the `pickle` data format uses a printable ASCII representation. This is slightly more voluminous than a binary representation. The big advantage of using printable ASCII (and of some other characteristics of `pickle`'s representation) is that for debugging or recovery purposes it is possible for a human to read the pickled file with a standard text editor.

There are currently 3 different protocols which can be used for pickling.

- Protocol version 0 is the original ASCII protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is the old binary format which is also compatible with earlier versions of Python.
- Protocol version 2 was introduced in Python 2.3. It provides much more efficient pickling of new-style classes.

Refer to **PEP 307** for more information.

If a *protocol* is not specified, protocol 0 is used. If *protocol* is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version available will be used.

Changed in version 2.3: Introduced the *protocol* parameter.

A binary format, which is slightly more efficient, can be chosen by specifying a *protocol* version >= 1.

11.1.3. Usage

To serialize an object hierarchy, you first create a pickler, then you call the pickler's `dump()` method. To de-serialize a data stream, you first create an unpickler, then you call the unpickler's `load()` method. The `pickle` module provides the following constant:

`pickle.`**`HIGHEST_PROTOCOL`**

The highest protocol version available. This value can be passed as a *protocol* value.

New in version 2.3.

Note: Be sure to always open pickle files created with protocols >= 1 in binary mode. For the old ASCII-based pickle protocol 0 you can use either text mode or binary mode as long as you stay consistent.

A pickle file written with protocol 0 in binary mode will contain lone linefeeds as line terminators and therefore will look "funny" when viewed in Notepad or other editors which do not support this format.

The `pickle` module provides the following functions to make the pickling process more convenient:

`pickle.`**`dump`**(*obj*, *file*[, *protocol*])

Write a pickled representation of *obj* to the open file object *file*. This is equivalent to `Pickler(file, protocol).dump(obj)`.

If the *protocol* parameter is omitted, protocol 0 is used. If *protocol* is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version will be used.

Changed in version 2.3: Introduced the *protocol* parameter.

file must have a `write()` method that accepts a single string argument. It can thus be a file object opened for writing, a `StringIO` object, or any other custom object that meets this interface.

`pickle.`**`load`**(*file*)

Read a string from the open file object *file* and interpret it as a pickle data stream, reconstructing and returning the original object hierarchy. This is equivalent to `Unpickler(file).load()`.

file must have two methods, a `read()` method that takes an integer argument, and a `readline()` method that requires no arguments. Both methods should return a string. Thus *file* can be a file object opened for reading, a `StringIO` object, or any other custom object that meets this interface.

This function automatically determines whether the data stream was written in binary mode or not.

`pickle.`**`dumps`**(*obj*[, *protocol*])

Return the pickled representation of the object as a string, instead of writing it to a file.

If the *protocol* parameter is omitted, protocol 0 is used. If *protocol* is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version will be used.

Changed in version 2.3: The *protocol* parameter was added.

`pickle.`**`loads`**(*string*)

Read a pickled object hierarchy from a string. Characters in the string past the pickled object's representation are ignored.

The `pickle` module also defines three exceptions:

exception `pickle.`**`PickleError`**

A common base class for the other exceptions defined below. This inherits from `Exception`.

exception `pickle.`**`PicklingError`**

This exception is raised when an unpicklable object is passed to the `dump()` method.

exception `pickle.`**`UnpicklingError`**

This exception is raised when there is a problem unpickling an object. Note that other exceptions may also be raised during unpickling, including (but not necessarily limited to) `AttributeError`, `EOFError`, `ImportError`, and `IndexError`.

The `pickle` module also exports two callables [2], `Pickler` and `Unpickler`:

class `pickle.`**`Pickler`**(*file*[, *protocol*])

This takes a file-like object to which it will write a pickle data stream.

If the *protocol* parameter is omitted, protocol 0 is used. If *protocol* is specified as a negative value or `HIGHEST_PROTOCOL`, the highest protocol version will be used.

Changed in version 2.3: Introduced the *protocol* parameter.

file must have a `write()` method that accepts a single string argument. It can thus be an open file object, a `StringIO` object, or any other custom object that meets this interface.

`Pickler` objects define one (or two) public methods:

`dump`(*obj*)

Write a pickled representation of *obj* to the open file object given in the constructor. Either the binary or ASCII format will be used, depending on the value of the *protocol* argument passed to the constructor.

clear_memo()

Clears the pickler's "memo". The memo is the data structure that remembers which objects the pickler has already seen, so that shared or recursive objects pickled by reference and not by value. This method is useful when re-using picklers.

> **Note:** Prior to Python 2.3, `clear_memo()` was only available on the picklers created by `cPickle`. In the `pickle` module, picklers have an instance variable called `memo` which is a Python dictionary. So to clear the memo for a `pickle` module pickler, you could do the following:
>
> ```
> mypickler.memo.clear()
> ```
>
> Code that does not need to support older versions of Python should simply use `clear_memo()`.

It is possible to make multiple calls to the `dump()` method of the same `Pickler` instance. These must then be matched to the same number of calls to the `load()` method of the corresponding `Unpickler` instance. If the same object is pickled by multiple `dump()` calls, the `load()` will all yield references to the same object. [3]

`Unpickler` objects are defined as:

class `pickle.`**Unpickler**(*file*)

This takes a file-like object from which it will read a pickle data stream. This class automatically determines whether the data stream was written in binary mode or not, so it does not need a flag as in the `Pickler` factory.

file must have two methods, a `read()` method that takes an integer argument, and a `readline()` method that requires no arguments. Both methods should return a string. Thus *file* can be a file object opened for reading, a `StringIO` object, or any other custom object that meets this interface.

`Unpickler` objects have one (or two) public methods:

load()

Read a pickled object representation from the open file object given in the constructor, and return the reconstituted object hierarchy specified therein.

This method automatically determines whether the data stream was written in binary mode or not.

noload()

This is just like `load()` except that it doesn't actually create any objects. This is useful primarily for finding what's called "persistent ids" that may be referenced in a pickle data stream. See section The pickle protocol below for more details.

Note: the `noload()` method is currently only available on `Unpickler` objects created with the `cPickle` module. `pickle` module `Unpickler`s do not have the

`noload()` method.

11.1.4. What can be pickled and unpickled?

The following types can be pickled:

- `None`, `True`, and `False`
- integers, long integers, floating point numbers, complex numbers
- normal and Unicode strings
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module
- built-in functions defined at the top level of a module
- classes that are defined at the top level of a module
- instances of such classes whose `__dict__` or the result of calling `__getstate__()` is picklable (see section The pickle protocol for details).

Attempts to pickle unpicklable objects will raise the `PicklingError` exception; when this happens, an unspecified number of bytes may have already been written to the underlying file. Trying to pickle a highly recursive data structure may exceed the maximum recursion depth, a `RuntimeError` will be raised in this case. You can carefully raise this limit with `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by "fully qualified" name reference, not by value. This means that only the function name is pickled, along with the name of the module the function is defined in. Neither the function's code, nor any of its function attributes are pickled. Thus the defining module must be importable in the unpickling environment, and the module must contain the named object, otherwise an exception will be raised. [4]

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 is pickled, so in the following example the class attribute `attr` is not restored in the unpickling environment:

```
class Foo:
    attr = 'a class attr'

picklestring = pickle.dumps(Foo)
```

These restrictions are why picklable functions and classes must be defined in the top level of a module.

Similarly, when class instances are pickled, their class's code and data are not pickled along with them. Only the instance 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. If you plan to have long-lived objects that will see many versions of a class, it may be worthwhile to put a version number in the objects so that suitable conversions can be made by the class's `__setstate__()` method.

11.1.5. The pickle protocol

This section describes the “pickling protocol” that defines the interface between the pickler/unpickler and the objects that are being serialized. This protocol provides a standard way for you to define, customize, and control how your objects are serialized and de-serialized. The description in this section doesn’t cover specific customizations that you can employ to make the unpickling environment slightly safer from untrusted pickle data streams; see section Subclassing Unpicklers for more details.

11.1.5.1. Pickling and unpickling normal class instances

`object.__getinitargs__()`

When a pickled class instance is unpickled, its `__init__()` method is normally *not* invoked. If it is desirable that the `__init__()` method be called on unpickling, an old-style class can define a method `__getinitargs__()`, which should return a *tuple* containing the arguments to be passed to the class constructor (`__init__()` for example). The `__getinitargs__()` method is called at pickle time; the tuple it returns is incorporated in the pickle for the instance.

`object.__getnewargs__()`

New-style types can provide a `__getnewargs__()` method that is used for protocol 2. Implementing this method is needed if the type establishes some internal invariants when the instance is created, or if the memory allocation is affected by the values passed to the `__new__()` method for the type (as it is for tuples and strings). Instances of a new-style class `c` are created using

```
obj = C.__new__(C, *args)
```

where *args* is the result of calling `__getnewargs__()` on the original object; if there is no `__getnewargs__()`, an empty tuple is assumed.

`object.__getstate__()`

Classes can further influence how their instances are pickled; if the class defines the method `__getstate__()`, it is called and the return state is pickled as the contents for the instance, instead of the contents of the instance’s dictionary. If there is no `__getstate__()` method, the instance’s `__dict__` is pickled.

`object.__setstate__(`*state*`)`

Upon unpickling, if the class also defines the method `__setstate__()`, it is called with the unpickled state. [5] If there is no `__setstate__()` method, the pickled state must be a dictionary and its items are assigned to the new instance’s dictionary. If a class defines both `__getstate__()` and `__setstate__()`, the state object needn’t be a dictionary and these methods can do what they want. [6]

> **Note:** For new-style classes, if `__getstate__()` returns a false value, the `__setstate__()` method will not be called.

Note: At unpickling time, some methods like `__getattr__()`, `__getattribute__()`, or `__setattr__()` may be called upon the instance. In case those methods rely on some internal invariant being true, the type should implement either `__getinitargs__()` or `__getnewargs__()` to establish such an invariant; otherwise, neither `__new__()` nor `__init__()` will be called.

11.1.5.2. Pickling and unpickling extension types

`object.__reduce__()`

When the `Pickler` encounters an object of a type it knows nothing about — such as an extension type — it looks in two places for a hint of how to pickle it. One alternative is for the object to implement a `__reduce__()` method. If provided, at pickling time `__reduce__()` will be called with no arguments, and it must return either a string or a tuple.

If a string is returned, it names a global variable whose contents are pickled as normal. The string returned by `__reduce__()` should be the object's local name relative to its module; the pickle module searches the module namespace to determine the object's module.

When a tuple is returned, it must be between two and five elements long. Optional elements can either be omitted, or `None` can be provided as their value. The contents of this tuple are pickled as normal and used to reconstruct the object at unpickling time. The semantics of each element are:

- A callable object that will be called to create the initial version of the object. The next element of the tuple will provide arguments for this callable, and later elements provide additional state information that will subsequently be used to fully reconstruct the pickled data.

  In the unpickling environment this object must be either a class, a callable registered as a "safe constructor" (see below), or it must have an attribute `__safe_for_unpickling__` with a true value. Otherwise, an `UnpicklingError` will be raised in the unpickling environment. Note that as usual, the callable itself is pickled by name.

- A tuple of arguments for the callable object.

  Changed in version 2.5: Formerly, this argument could also be `None`.

- Optionally, the object's state, which will be passed to the object's `__setstate__()` method as described in section Pickling and unpickling normal class instances. If the object has no `__setstate__()` method, then, as above, the value must be a dictionary and it will be added to the object's `__dict__`.

- Optionally, an iterator (and not a sequence) yielding successive list items. These list items will be pickled, and appended to the object using either `obj.append(item)` or `obj.extend(list_of_items)`. This is primarily used for list

subclasses, but may be used by other classes as long as they have **append()** and **extend()** methods with the appropriate signature. (Whether **append()** or **extend()** is used depends on which pickle protocol version is used as well as the number of items to append, so both must be supported.)

- Optionally, an iterator (not a sequence) yielding successive dictionary items, which should be tuples of the form `(key, value)`. These items will be pickled and stored to the object using `obj[key] = value`. This is primarily used for dictionary subclasses, but may be used by other classes as long as they implement **__setitem__()**.

`object.`**__reduce_ex__**(*protocol*)

It is sometimes useful to know the protocol version when implementing **__reduce__()**. This can be done by implementing a method named **__reduce_ex__()** instead of **__reduce__()**. **__reduce_ex__()**, when it exists, is called in preference over **__reduce__()** (you may still provide **__reduce__()** for backwards compatibility). The **__reduce_ex__()** method will be called with a single integer argument, the protocol version.

The **object** class implements both **__reduce__()** and **__reduce_ex__()**; however, if a subclass overrides **__reduce__()** but not **__reduce_ex__()**, the **__reduce_ex__()** implementation detects this and calls **__reduce__()**.

An alternative to implementing a **__reduce__()** method on the object to be pickled, is to register the callable with the **copy_reg** module. This module provides a way for programs to register "reduction functions" and constructors for user-defined types. Reduction functions have the same semantics and interface as the **__reduce__()** method described above, except that they are called with a single argument, the object to be pickled.

The registered constructor is deemed a "safe constructor" for purposes of unpickling as described above.

11.1.5.3. Pickling and unpickling external objects

For the benefit of object persistence, the **pickle** module supports the notion of a reference to an object outside the pickled data stream. Such objects are referenced by a "persistent id", which is just an arbitrary string of printable ASCII characters. The resolution of such names is not defined by the **pickle** module; it will delegate this resolution to user defined functions on the pickler and unpickler. [7]

To define external persistent id resolution, you need to set the **persistent_id** attribute of the pickler object and the **persistent_load** attribute of the unpickler object.

To pickle objects that have an external persistent id, the pickler must have a custom **persistent_id()** method that takes an object as an argument and returns either `None` or the persistent id for that object. When `None` is returned, the pickler simply pickles the object as normal. When a persistent id string is returned, the pickler will pickle that string, along with a marker so that the unpickler will recognize the string as a persistent id.

To unpickle external objects, the unpickler must have a custom **persistent_load()** function that takes a persistent id string and returns the referenced object.

Here's a silly example that *might* shed more light:

```
import pickle
from cStringIO import StringIO

src = StringIO()
p = pickle.Pickler(src)

def persistent_id(obj):
    if hasattr(obj, 'x'):
        return 'the value %d' % obj.x
    else:
        return None

p.persistent_id = persistent_id

class Integer:
    def __init__(self, x):
        self.x = x
    def __str__(self):
        return 'My name is integer %d' % self.x

i = Integer(7)
print i
p.dump(i)

datastream = src.getvalue()
print repr(datastream)
dst = StringIO(datastream)

up = pickle.Unpickler(dst)

class FancyInteger(Integer):
    def __str__(self):
        return 'I am the integer %d' % self.x

def persistent_load(persid):
    if persid.startswith('the value '):
        value = int(persid.split()[2])
        return FancyInteger(value)
    else:
        raise pickle.UnpicklingError, 'Invalid persistent id'

up.persistent_load = persistent_load

j = up.load()
print j
```

In the **cPickle** module, the unpickler's **persistent_load** attribute can also be set to a Python list, in which case, when the unpickler reaches a persistent id, the persistent id string will simply be appended to this list. This functionality exists so that a pickle data stream can be "sniffed" for object references without actually instantiating all the objects in a pickle. [8] Setting **persistent_load** to a list is usually used in conjunction with the **noload()** method on the Unpickler.

11.1.6. Subclassing Unpicklers

By default, unpickling will import any class that it finds in the pickle data. You can control exactly what gets unpickled and what gets called by customizing your unpickler. Unfortunately, exactly how you do this is different depending on whether you're using **pickle** or **cPickle**. [9]

In the **pickle** module, you need to derive a subclass from **Unpickler**, overriding the **load_global()** method. **load_global()** should read two lines from the pickle data stream where the first line will the name of the module containing the class and the second line will be the name of the instance's class. It then looks up the class, possibly importing the module and digging out the attribute, then it appends what it finds to the unpickler's stack. Later on, this class will be assigned to the **__class__** attribute of an empty class, as a way of magically creating an instance without calling its class's **__init__()**. Your job (should you choose to accept it), would be to have **load_global()** push onto the unpickler's stack, a known safe version of any class you deem safe to unpickle. It is up to you to produce such a class. Or you could raise an error if you want to disallow all unpickling of instances. If this sounds like a hack, you're right. Refer to the source code to make this work.

Things are a little cleaner with **cPickle**, but not by much. To control what gets unpickled, you can set the unpickler's **find_global** attribute to a function or `None`. If it is `None` then any attempts to unpickle instances will raise an **UnpicklingError**. If it is a function, then it should accept a module name and a class name, and return the corresponding class object. It is responsible for looking up the class and performing any necessary imports, and it may raise an error to prevent instances of the class from being unpickled.

The moral of the story is that you should be really careful about the source of the strings your application unpickles.

11.1.7. Example

For the simplest code, use the **dump()** and **load()** functions. Note that a self-referencing list is pickled and restored correctly.

```
import pickle

data1 = {'a': [1, 2.0, 3, 4+6j],
         'b': ('string', u'Unicode string'),
         'c': None}

selfref_list = [1, 2, 3]
selfref_list.append(selfref_list)

output = open('data.pkl', 'wb')

# Pickle dictionary using protocol 0.
pickle.dump(data1, output)

# Pickle the list using the highest protocol available.
pickle.dump(selfref_list, output, -1)

output.close()
```

The following example reads the resulting pickled data. When reading a pickle-containing file, you should open the file in binary mode because you can't be sure if the ASCII or binary format was used.

```
import pprint, pickle

pkl_file = open('data.pkl', 'rb')

data1 = pickle.load(pkl_file)
pprint.pprint(data1)

data2 = pickle.load(pkl_file)
pprint.pprint(data2)

pkl_file.close()
```

Here's a larger example that shows how to modify pickling behavior for a class. The **TextReader** class opens a text file, and returns the line number and line contents each time its **readline()** method is called. If a **TextReader** instance is pickled, all attributes *except* the file object member are saved. When the instance is unpickled, the file is reopened, and reading resumes from the last location. The **__setstate__()** and **__getstate__()** methods are used to implement this behavior.

```
#!/usr/local/bin/python

class TextReader:
    """Print and number lines in a text file."""
    def __init__(self, file):
        self.file = file
        self.fh = open(file)
        self.lineno = 0

    def readline(self):
        self.lineno = self.lineno + 1
        line = self.fh.readline()
        if not line:
            return None
        if line.endswith("\n"):
            line = line[:-1]
        return "%d: %s" % (self.lineno, line)

    def __getstate__(self):
        odict = self.__dict__.copy() # copy the dict since we change it
        del odict['fh']              # remove filehandle entry
        return odict

    def __setstate__(self, dict):
        fh = open(dict['file'])      # reopen file
        count = dict['lineno']       # read from file...
        while count:                 # until line count is restored
            fh.readline()
            count = count - 1
        self.__dict__.update(dict)   # update attributes
        self.fh = fh                 # save the file object
```

A sample usage might be something like this:

```
>>> import TextReader
>>> obj = TextReader.TextReader("TextReader.py")
```

```
>>> obj.readline()
'1: #!/usr/local/bin/python'
>>> obj.readline()
'2: '
>>> obj.readline()
'3: class TextReader:'
>>> import pickle
>>> pickle.dump(obj, open('save.p', 'wb'))
```

If you want to see that `pickle` works across Python processes, start another Python session, before continuing. What follows can happen from either the same process or a new process.

```
>>> import pickle
>>> reader = pickle.load(open('save.p', 'rb'))
>>> reader.readline()
'4:     """Print and number lines in a text file."""'
```

See also:

Module `copy_reg`
Pickle interface constructor registration for extension types.

Module `shelve`
Indexed databases of objects; uses `pickle`.

Module `copy`
Shallow and deep object copying.

Module `marshal`
High-performance serialization of built-in types.

11.2. `cPickle` — A faster `pickle`

The `cPickle` module supports serialization and de-serialization of Python objects, providing an interface and functionality nearly identical to the `pickle` module. There are several differences, the most important being performance and subclassability.

First, `cPickle` can be up to 1000 times faster than `pickle` because the former is implemented in C. Second, in the `cPickle` module the callables `Pickler()` and `Unpickler()` are functions, not classes. This means that you cannot use them to derive custom pickling and unpickling subclasses. Most applications have no need for this functionality and should benefit from the greatly improved performance of the `cPickle` module.

The pickle data stream produced by `pickle` and `cPickle` are identical, so it is possible to use `pickle` and `cPickle` interchangeably with existing pickles. [10]

There are additional minor differences in API between `cPickle` and `pickle`, however for most applications, they are interchangeable. More documentation is provided in the `pickle` module documentation, which includes a list of the documented differences.

Footnotes

[1] Don't confuse this with the `marshal` module

[2] In the `pickle` module these callables are classes, which you could subclass to customize the behavior. However, in the `cPickle` module these callables are factory functions and so cannot be subclassed. One common reason to subclass is to control what objects can actually be unpickled. See section Subclassing Unpicklers for more details.

[3] *Warning*: this is intended for pickling multiple objects without intervening modifications to the objects or their parts. If you modify an object and then pickle it again using the same `Pickler` instance, the object is not pickled again — a reference to it is pickled and the `Unpickler` will return the old value, not the modified one. There are two problems here: (1) detecting changes, and (2) marshalling a minimal set of changes. Garbage Collection may also become a problem here.

[4] The exception raised will likely be an `ImportError` or an `AttributeError` but it could be something else.

[5] These methods can also be used to implement copying class instances.

[6] This protocol is also used by the shallow and deep copying operations defined in the `copy` module.

[7] The actual mechanism for associating these user defined functions is slightly different for `pickle` and `cPickle`. The description given here works the same for both implementations. Users of the `pickle` module could also use subclassing to effect the same results, overriding the `persistent_id()` and `persistent_load()` methods in the derived classes.

[8] We'll leave you with the image of Guido and Jim sitting around sniffing pickles in their living rooms.

[9] A word of caution: the mechanisms described here use internal attributes and methods, which are subject to change in future versions of Python. We intend to someday provide a common interface for controlling this behavior, which will work in either `pickle` or `cPickle`.

[10] Since the pickle data format is actually a tiny stack-oriented programming language, and some freedom is taken in the encodings of certain objects, it is possible that the two modules produce different data streams for the same input objects. However it is guaranteed that they will always be able to read each other's data streams.