All your code in one place

Dismiss

GitHub makes it easy to scale back on context switching. Read rendered documentation, see the history of any file, and collaborate with contributors on projects across GitHub.

Sign up for free   See pricing for teams and enterprises

dimaqq ~~bpo-38388~~: Document pickle protocol version 5 (GH-16639)

d0e0f5b   19 days ago

30 contributors   and others

:mod:`pickle` --- Python object serialization

```
.. module:: pickle
   :synopsis: Convert Python objects to streams of bytes and back.
```

```
.. sectionauthor:: Jim Kerr <jbkerr@sr.hp.com>.
```

```
.. sectionauthor:: Barry Warsaw <barry@python.org>
```

Source code: :source:`Lib/pickle.py`

```
.. index::
   single: persistence
   pair: persistent; objects
   pair: serializing; objects
```

```
    pair: marshalling; objects
    pair: flattening; objects
    pair: pickling; objects
```

The :mod:`pickle` module implements binary protocols 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 (from a :term:`binary file` or :term:`bytes-like object`) 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".

Warning

The `pickle` module **is not secure**. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

Consider signing data with :mod:`hmac` if you need to ensure that it has not been tampered with.

Safer serialization formats such as :mod:`json` may be more appropriate if you are processing untrusted data. See :ref:`comparison-with-json`.

Relationship to other Python modules

Comparison with `marshal`

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

The :mod:`pickle` module differs from :mod:`marshal` in several significant ways:

- The :mod:`pickle` module keeps track of the objects it has already serialized, so that later references to the same object won't be serialized again. :mod:`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. :mod:`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.

- :mod:`marshal` cannot be used to serialize user-defined classes and their instances. :mod:`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 :mod:`marshal` serialization format is not guaranteed to be portable across Python versions. Because its primary job in life is to support :file:`.pyc` files, the Python implementers reserve the right to change the serialization format in non-backwards compatible ways should the need arise. The :mod:`pickle` serialization format is guaranteed to be backwards compatible across Python releases provided a compatible pickle protocol is chosen and pickling and unpickling code deals with Python 2 to Python 3 type differences if your data is crossing that unique breaking change language boundary.

Comparison with `json`

There are fundamental differences between the pickle protocols and ~~JSON (JavaScript Object Notation)~~:

- JSON is a text serialization format (it outputs unicode text, although most of the time it is then encoded to `utf-8`), while pickle is a binary serialization format;
- JSON is human-readable, while pickle is not;
- JSON is interoperable and widely used outside of the Python ecosystem, while pickle is Python-specific;
- JSON, by default, can only represent a subset of the Python built-in types, and no custom classes; pickle can represent an extremely large number of Python types (many of them automatically, by clever usage of Python's introspection facilities; complex cases can be tackled by implementing :ref:`specific object APIs <pickle-inst>`);
- Unlike pickle, deserializing untrusted JSON does not in itself create an arbitrary code execution vulnerability.

```
.. seealso::
   The :mod:`json` module: a standard library module allowing JSON
   serialization and deserialization.
```

Data stream format

```
.. index::
   single: External Data Representation
```

The data format used by :mod:`pickle` is Python-specific. This has the advantage that there are no restrictions imposed by external standards such as JSON or 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 :mod:`pickle` data format uses a relatively compact binary representation. If you need optimal size characteristics, you can efficiently :doc:`compress <archiving>` pickled data.

The module :mod:`pickletools` contains tools for analyzing data streams generated by :mod:`pickle`. :mod:`pickletools` source code has extensive comments about opcodes used by pickle protocols.

There are currently 6 different protocols which can be used for pickling. The higher the protocol used, the more recent the version of Python needed to read the pickle produced.

- Protocol version 0 is the original "human-readable" protocol and is backwards compatible with earlier versions of Python.
- Protocol version 1 is an 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 :term:`new-style class`es. Refer to ~~PEP 307~~ for information about improvements brought by protocol 2.
- Protocol version 3 was added in Python 3.0. It has explicit support for :class:`bytes` objects and cannot be unpickled by Python 2.x. This was the default protocol in Python 3.0--3.7.
- Protocol version 4 was added in Python 3.4. It adds support for very large objects, pickling more kinds of objects, and some data format optimizations. It is the default protocol starting with Python 3.8. Refer to ~~PEP 3154~~ for information about improvements brought by protocol 4.
- Protocol version 5 was added in Python 3.8. It adds support for out-of-band data and speedup for in-band data. Refer to ~~PEP 574~~ for information about improvements brought by protocol 5.

Note

Serialization is a more primitive notion than persistence; although :mod:`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 :mod:`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 :mod:`shelve` module provides a simple interface to pickle and unpickle objects on DBM-style database files.

Module Interface

To serialize an object hierarchy, you simply call the :func:`dumps` function. Similarly, to de-serialize a data stream, you call the :func:`loads` function. However, if you want more control over serialization and de-serialization, you can create a :class:`Pickler` or an :class:`Unpickler` object, respectively.

The :mod:`pickle` module provides the following constants:

```
.. data:: HIGHEST_PROTOCOL

   An integer, the highest :ref:`protocol version <pickle-protocols>`
   available.  This value can be passed as a *protocol* value to functions
   :func:`dump` and :func:`dumps` as well as the :class:`Pickler`
   constructor.
```

```
.. data:: DEFAULT_PROTOCOL

   An integer, the default :ref:`protocol version <pickle-protocols>` used
   for pickling.  May be less than :data:`HIGHEST_PROTOCOL`.  Currently the
   default protocol is 4, first introduced in Python 3.4 and incompatible
   with previous versions.

   .. versionchanged:: 3.0

      The default protocol is 3.

   .. versionchanged:: 3.8

      The default protocol is 4.
```

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

```
.. function:: dump(obj, file, protocol=None, \*, fix_imports=True, buffer_callback=None

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

   Arguments *file*, *protocol*, *fix_imports* and *buffer_callback* have
   the same meaning as in the :class:`Pickler` constructor.

   .. versionchanged:: 3.8
      The *buffer_callback* argument was added.
```

```
.. function:: dumps(obj, protocol=None, \*, fix_imports=True, buffer_callback=None)

   Return the pickled representation of the object *obj* as a :class:`bytes` object,
   instead of writing it to a file.

   Arguments *protocol*, *fix_imports* and *buffer_callback* have the same
   meaning as in the :class:`Pickler` constructor.

   .. versionchanged:: 3.8
      The *buffer_callback* argument was added.



.. function:: load(file, \*, fix_imports=True, encoding="ASCII", errors="strict", buffer

   Read the pickled representation of an object from the open :term:`file object`
   *file* and return the reconstituted object hierarchy specified therein.
   This is equivalent to ``Unpickler(file).load()``.

   The protocol version of the pickle is detected automatically, so no
   protocol argument is needed.  Bytes past the pickled representation
   of the object are ignored.

   Arguments *file*, *fix_imports*, *encoding*, *errors*, *strict* and *buffers*
   have the same meaning as in the :class:`Unpickler` constructor.

   .. versionchanged:: 3.8
      The *buffers* argument was added.
```

```
.. function:: loads(bytes_object, \*, fix_imports=True, encoding="ASCII", errors="strict

   Return the reconstituted object hierarchy of the pickled representation
   *bytes_object* of an object.

   The protocol version of the pickle is detected automatically, so no
   protocol argument is needed.  Bytes past the pickled representation
   of the object are ignored.

   Arguments *file*, *fix_imports*, *encoding*, *errors*, *strict* and *buffers*
   have the same meaning as in the :class:`Unpickler` constructor.

   .. versionchanged:: 3.8
      The *buffers* argument was added.
```

The :mod:`pickle` module defines three exceptions:

```
.. exception:: PickleError

   Common base class for the other pickling exceptions.  It inherits
   :exc:`Exception`.
```

```
.. exception:: PicklingError

   Error raised when an unpicklable object is encountered by :class:`Pickler`.
   It inherits :exc:`PickleError`.

   Refer to :ref:`pickle-picklable` to learn what kinds of objects can be
   pickled.
```

```
.. exception:: UnpicklingError

   Error raised when there is a problem unpickling an object, such as a data
   corruption or a security violation.  It inherits :exc:`PickleError`.

   Note that other exceptions may also be raised during unpickling, including
   (but not necessarily limited to) AttributeError, EOFError, ImportError, and
   IndexError.
```

The :mod:`pickle` module exports three classes, :class:`Pickler`, :class:`Unpickler` and :class:`PickleBuffer`:

A wrapper for a buffer representing picklable data. *buffer* must be a :ref:`buffer-providing <bufferobjects>` object, such as a :term:`bytes-like object` or a N-dimensional array.

:class:`PickleBuffer` is itself a buffer provider, therefore it is possible to pass it to other APIs expecting a buffer-providing object, such as :class:`memoryview`.

:class:`PickleBuffer` objects can only be serialized using pickle protocol 5 or higher. They are eligible for :ref:`out-of-band serialization <pickle-oob>`.

```
.. versionadded:: 3.8
```

```
.. method:: raw()

   Return a :class:`memoryview` of the memory area underlying this buffer.
   The returned object is a one-dimensional, C-contiguous memoryview
   with format ``B`` (unsigned bytes).  :exc:`BufferError` is raised if
   the buffer is neither C- nor Fortran-contiguous.
```

```
.. method:: release()

   Release the underlying buffer exposed by the PickleBuffer object.
```

What can be pickled and unpickled?

The following types can be pickled:

- `None`, `True`, and `False`
- integers, floating point numbers, complex numbers
- strings, bytes, bytearrays
- tuples, lists, sets, and dictionaries containing only picklable objects
- functions defined at the top level of a module (using :keyword:`def`, not :keyword:`lambda`)
- 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 :attr:`~object.__dict__` or the result of calling :meth:`__getstate__` is picklable (see section :ref:`pickle-inst` for details).

Attempts to pickle unpicklable objects will raise the :exc:`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 :exc:`RecursionError` will be raised in this case. You can carefully raise this limit with :func:`sys.setrecursionlimit`.

Note that functions (built-in and user-defined) are pickled by "fully qualified" name reference, not by value. [2] 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. [3]

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 attribute'

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 :meth:`__setstate__` method.

Pickling Class Instances

```
.. currentmodule:: None
```

In this section, we describe the general mechanisms available to you to define, customize, and control how class instances are pickled and unpickled.

In most cases, no additional code is needed to make instances picklable. By default, pickle will retrieve the class and the attributes of an instance via introspection. When a class instance is unpickled, its :meth:`__init__` method is usually *not* invoked. The default behaviour first creates an uninitialized instance and then restores the saved attributes. The following code shows an implementation of this behaviour:

```
def save(obj):
    return (obj.__class__, obj.__dict__)

def load(cls, attributes):
    obj = cls.__new__(cls)
    obj.__dict__.update(attributes)
    return obj
```

Classes can alter the default behaviour by providing one or several special methods:

```
.. method:: object.__getnewargs_ex__()

   In protocols 2 and newer, classes that implements the
   :meth:`__getnewargs_ex__` method can dictate the values passed to the
   :meth:`__new__` method upon unpickling.  The method must return a pair
   ``(args, kwargs)`` where *args* is a tuple of positional arguments
   and *kwargs* a dictionary of named arguments for constructing the
   object.  Those will be passed to the :meth:`__new__` method upon
   unpickling.

   You should implement this method if the :meth:`__new__` method of your
   class requires keyword-only arguments.  Otherwise, it is recommended for
   compatibility to implement :meth:`__getnewargs__`.

   .. versionchanged:: 3.6
      :meth:`__getnewargs_ex__` is now used in protocols 2 and 3.




.. method:: object.__getnewargs__()

   This method serves a similar purpose as :meth:`__getnewargs_ex__`, but
   supports only positional arguments.  It must return a tuple of arguments
   ``args`` which will be passed to the :meth:`__new__` method upon unpickling.

   :meth:`__getnewargs__` will not be called if :meth:`__getnewargs_ex__` is
   defined.

   .. versionchanged:: 3.6
      Before Python 3.6, :meth:`__getnewargs__` was called instead of
      :meth:`__getnewargs_ex__` in protocols 2 and 3.
```

```
.. method:: object.__getstate__()

   Classes can further influence how their instances are pickled; if the class
   defines the method :meth:`__getstate__`, it is called and the returned object
   is pickled as the contents for the instance, instead of the contents of the
   instance's dictionary.  If the :meth:`__getstate__` method is absent, the
   instance's :attr:`~object.__dict__` is pickled as usual.
```

```
.. method:: object.__setstate__(state)

   Upon unpickling, if the class defines :meth:`__setstate__`, it is called with
   the unpickled state.  In that case, there is no requirement for the state
   object to be a dictionary.  Otherwise, the pickled state must be a dictionary
   and its items are assigned to the new instance's dictionary.

   .. note::

      If :meth:`__getstate__` returns a false value, the :meth:`__setstate__`
      method will not be called upon unpickling.
```

Refer to the section :ref:`pickle-state` for more information about how to use the methods :meth:`__getstate__` and :meth:`__setstate__`.

Note

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

```
.. index:: pair: copy; protocol
```

As we shall see, pickle does not use directly the methods described above. In fact, these methods are part of the copy protocol which implements the :meth:`__reduce__` special method. The copy protocol provides a unified interface for retrieving the data necessary for pickling and copying objects. [4]

Although powerful, implementing :meth:`__reduce__` directly in your classes is error prone. For this reason, class designers should use the high-level interface (i.e., :meth:`__getnewargs_ex__`, :meth:`__getstate__` and :meth:`__setstate__`) whenever possible. We will show, however, cases where using :meth:`__reduce__` is the only option or leads to more efficient pickling or both.

```
.. method:: object.__reduce__()

   The interface is currently defined as follows.  The :meth:`__reduce__` method
   takes no argument and shall return either a string or preferably a tuple (the
   returned object is often referred to as the "reduce value").

   If a string is returned, the string should be interpreted as the name of a
   global variable.  It should be the object's local name relative to its
   module; the pickle module searches the module namespace to determine the
   object's module.  This behaviour is typically useful for singletons.

   When a tuple is returned, it must be between two and six items long.
   Optional items can either be omitted, or ``None`` can be provided as their
   value.  The semantics of each item are in order:

   .. XXX Mention __newobj__ special-case?

   * A callable object that will be called to create the initial version of the
     object.

   * A tuple of arguments for the callable object.  An empty tuple must be given
     if the callable does not accept any argument.

   * Optionally, the object's state, which will be passed to the object's
     :meth:`__setstate__` method as previously described.  If the object has no
     such method then, the value must be a dictionary and it will be added to
     the object's :attr:`~object.__dict__` attribute.

   * Optionally, an iterator (and not a sequence) yielding successive items.
     These items will be appended to the object either using
     ``obj.append(item)`` or, in batch, using ``obj.extend(list_of_items)``.
     This is primarily used for list subclasses, but may be used by other
     classes as long as they have :meth:`append` and :meth:`extend` methods with
     the appropriate signature.  (Whether :meth:`append` or :meth:`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 key-value
     pairs.  These items will be 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 :meth:`__setitem__`.

   * Optionally, a callable with a ``(obj, state)`` signature. This
     callable allows the user to programmatically control the state-updating
     behavior of a specific object, instead of using ``obj``'s static
     :meth:`__setstate__` method. If not ``None``, this callable will have
     priority over ``obj``'s :meth:`__setstate__`.

     .. versionadded:: 3.8
        The optional sixth tuple item, ``(obj, state)``, was added.
```

```
.. method:: object.__reduce_ex__(protocol)

   Alternatively, a :meth:`__reduce_ex__` method may be defined.  The only
   difference is this method should take a single integer argument, the protocol
   version.  When defined, pickle will prefer it over the :meth:`__reduce__`
   method.  In addition, :meth:`__reduce__` automatically becomes a synonym for
   the extended version.  The main use for this method is to provide
   backwards-compatible reduce values for older Python releases.
```

```
.. currentmodule:: pickle
```

Persistence of External Objects

```
.. index::
   single: persistent_id (pickle protocol)
   single: persistent_load (pickle protocol)
```

For the benefit of object persistence, the :mod:`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 should be either a string of alphanumeric characters (for protocol 0) [5] or just an arbitrary object (for any newer protocol).

The resolution of such persistent IDs is not defined by the :mod:`pickle` module; it will delegate this resolution to the user-defined methods on the pickler and unpickler, :meth:`~Pickler.persistent_id` and :meth:`~Unpickler.persistent_load` respectively.

To pickle objects that have an external persistent ID, the pickler must have a custom :meth:`~Pickler.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 object, along with a marker so that the unpickler will recognize it as a persistent ID.

To unpickle external objects, the unpickler must have a custom :meth:`~Unpickler.persistent_load` method that takes a persistent ID object and returns the referenced object.

Here is a comprehensive example presenting how persistent ID can be used to pickle external objects by reference.

```
.. literalinclude:: ../includes/dbpickle.py
```

Dispatch Tables

If one wants to customize pickling of some classes without disturbing any other code which depends on pickling, then one can create a pickler with a private dispatch table.

The global dispatch table managed by the :mod:`copyreg` module is available as :data:`copyreg.dispatch_table`. Therefore, one may choose to use a modified copy of :data:`copyreg.dispatch_table` as a private dispatch table.

For example

```
f = io.BytesIO()
p = pickle.Pickler(f)
p.dispatch_table = copyreg.dispatch_table.copy()
p.dispatch_table[SomeClass] = reduce_SomeClass
```

creates an instance of :class:`pickle.Pickler` with a private dispatch table which handles the `SomeClass` class specially. Alternatively, the code

```
class MyPickler(pickle.Pickler):
    dispatch_table = copyreg.dispatch_table.copy()
    dispatch_table[SomeClass] = reduce_SomeClass
f = io.BytesIO()
p = MyPickler(f)
```

does the same, but all instances of `MyPickler` will by default share the same dispatch table. The equivalent code using the :mod:`copyreg` module is

```
copyreg.pickle(SomeClass, reduce_SomeClass)
f = io.BytesIO()
p = pickle.Pickler(f)
```

Handling Stateful Objects

```
.. index::
   single: __getstate__() (copy protocol)
   single: __setstate__() (copy protocol)
```

Here's an example that shows how to modify pickling behavior for a class. The :class:`TextReader` class opens a text file, and returns the line number and line contents each time its :meth:`!readline` method is called. If a :class:`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 :meth:`__setstate__` and :meth:`__getstate__` methods are used to implement this behavior.

```
class TextReader:
    """Print and number lines in a text file."""

    def __init__(self, filename):
        self.filename = filename
        self.file = open(filename)
        self.lineno = 0

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

    def __getstate__(self):
        # Copy the object's state from self.__dict__ which contains
        # all our instance attributes. Always use the dict.copy()
        # method to avoid modifying the original state.
        state = self.__dict__.copy()
        # Remove the unpicklable entries.
        del state['file']
        return state

    def __setstate__(self, state):
        # Restore instance attributes (i.e., filename and lineno).
        self.__dict__.update(state)
        # Restore the previously opened file's state. To do so, we need to
        # reopen it and read from it until the line count is restored.
        file = open(self.filename)
        for _ in range(self.lineno):
            file.readline()
        # Finally, save the file.
        self.file = file
```

A sample usage might be something like this:

```
>>> reader = TextReader("hello.txt")
>>> reader.readline()
'1: Hello world!'
>>> reader.readline()
'2: I am line number two.'
>>> new_reader = pickle.loads(pickle.dumps(reader))
>>> new_reader.readline()
'3: Goodbye!'
```

Custom Reduction for Types, Functions, and Other Objects

```
.. versionadded:: 3.8
```

Sometimes, :attr:`~Pickler.dispatch_table` may not be flexible enough. In particular we may want to customize pickling based on another criterion than the object's type, or we may want to customize the pickling of functions and classes.

For those cases, it is possible to subclass from the :class:`Pickler` class and implement a :meth:`~Pickler.reducer_override` method. This method can return an arbitrary reduction tuple (see :meth:`__reduce__`). It can alternatively return `NotImplemented` to fallback to the traditional behavior.

If both the :attr:`~Pickler.dispatch_table` and :meth:`~Pickler.reducer_override` are defined, then :meth:`~Pickler.reducer_override` method takes priority.

Note

For performance reasons, :meth:`~Pickler.reducer_override` may not be called for the following objects: `None` , `True` , `False` , and exact instances of :class:`int`, :class:`float`, :class:`bytes`, :class:`str`, :class:`dict`, :class:`set`, :class:`frozenset`, :class:`list` and :class:`tuple`.

Here is a simple example where we allow pickling and reconstructing a given class:

```
import io
import pickle

class MyClass:
    my_attribute = 1

class MyPickler(pickle.Pickler):
    def reducer_override(self, obj):
        """Custom reducer for MyClass."""
        if getattr(obj, "__name__", None) == "MyClass":
            return type, (obj.__name__, obj.__bases__,
                          {'my_attribute': obj.my_attribute})
        else:
            # For any other object, fallback to usual reduction
            return NotImplemented

f = io.BytesIO()
p = MyPickler(f)
p.dump(MyClass)

del MyClass

unpickled_class = pickle.loads(f.getvalue())

assert isinstance(unpickled_class, type)
assert unpickled_class.__name__ == "MyClass"
assert unpickled_class.my_attribute == 1
```

Out-of-band Buffers

```
.. versionadded:: 3.8
```

In some contexts, the :mod:`pickle` module is used to transfer massive amounts of data. Therefore, it can be important to minimize the number of memory copies, to preserve performance and resource consumption. However, normal operation of the :mod:`pickle` module, as it transforms a graph-like structure of objects into a sequential stream of bytes, intrinsically involves copying data to and from the pickle stream.

This constraint can be eschewed if both the *provider* (the implementation of the object types to be transferred) and the *consumer* (the implementation of the communications system) support the out-of-band transfer facilities provided by pickle protocol 5 and higher.

Provider API

The large data objects to be pickled must implement a :meth:`__reduce_ex__` method specialized for protocol 5 and higher, which returns a :class:`PickleBuffer` instance (instead of e.g. a :class:`bytes` object) for any large data.

A :class:`PickleBuffer` object *signals* that the underlying buffer is eligible for out-of-band data transfer. Those objects remain compatible with normal usage of the :mod:`pickle` module. However, consumers can also opt-in to tell :mod:`pickle` that they will handle those buffers by themselves.

Consumer API

A communications system can enable custom handling of the :class:`PickleBuffer` objects generated when serializing an object graph.

On the sending side, it needs to pass a *buffer_callback* argument to :class:`Pickler` (or to the :func:`dump` or :func:`dumps` function), which will be called with each :class:`PickleBuffer` generated while pickling the object graph. Buffers accumulated by the *buffer_callback* will not see their data copied into the pickle stream, only a cheap marker will be inserted.

On the receiving side, it needs to pass a *buffers* argument to :class:`Unpickler` (or to the :func:`load` or :func:`loads` function), which is an iterable of the buffers which were passed to *buffer_callback*. That iterable should produce buffers in the same order as they were passed to *buffer_callback*. Those buffers will provide the data expected by the reconstructors of the objects whose pickling produced the original :class:`PickleBuffer` objects.

Between the sending side and the receiving side, the communications system is free to implement its own transfer mechanism for out-of-band buffers. Potential optimizations include the use of shared memory or datatype-dependent compression.

Example

Here is a trivial example where we implement a :class:`bytearray` subclass able to participate in out-of-band buffer pickling:

```
class ZeroCopyByteArray(bytearray):

    def __reduce_ex__(self, protocol):
        if protocol >= 5:
            return type(self)._reconstruct, (PickleBuffer(self),), None
        else:
            # PickleBuffer is forbidden with pickle protocols <= 4.
            return type(self)._reconstruct, (bytearray(self),)

    @classmethod
    def _reconstruct(cls, obj):
        with memoryview(obj) as m:
            # Get a handle over the original buffer object
            obj = m.obj
            if type(obj) is cls:
                # Original buffer object is a ZeroCopyByteArray, return it
                # as-is.
                return obj
            else:
                return cls(obj)
```

The reconstructor (the `_reconstruct` class method) returns the buffer's providing object if it has the right type. This is an easy way to simulate zero-copy behaviour on this toy example.

On the consumer side, we can pickle those objects the usual way, which when unserialized will give us a copy of the original object:

```
b = ZeroCopyByteArray(b"abc")
data = pickle.dumps(b, protocol=5)
new_b = pickle.loads(data)
print(b == new_b)  # True
print(b is new_b)  # False: a copy was made
```

But if we pass a *buffer_callback* and then give back the accumulated buffers when unserializing, we are able to get back the original object:

```
b = ZeroCopyByteArray(b"abc")
buffers = []
data = pickle.dumps(b, protocol=5, buffer_callback=buffers.append)
new_b = pickle.loads(data, buffers=buffers)
print(b == new_b)  # True
print(b is new_b)  # True: no copy was made
```

This example is limited by the fact that :class:`bytearray` allocates its own memory: you cannot create a :class:`bytearray` instance that is backed by another object's memory. However, third-party datatypes such as NumPy arrays do not have this limitation, and allow use of zero-copy pickling (or making as few copies as possible) when transferring between distinct processes or systems.

```
.. seealso:: :pep:`574` -- Pickle protocol 5 with out-of-band data
```

Restricting Globals

```
.. index::
   single: find_class() (pickle protocol)
```

By default, unpickling will import any class or function that it finds in the pickle data. For many applications, this behaviour is unacceptable as it permits the unpickler to import and invoke arbitrary code. Just consider what this hand-crafted pickle data stream does when loaded:

```
>>> import pickle
>>> pickle.loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
hello world
0
```

In this example, the unpickler imports the :func:`os.system` function and then apply the string argument "echo hello world". Although this example is inoffensive, it is not difficult to imagine one that could damage your system.

For this reason, you may want to control what gets unpickled by customizing :meth:`Unpickler.find_class`. Unlike its name suggests, :meth:`Unpickler.find_class` is called whenever a global (i.e., a class or a function) is requested. Thus it is possible to either completely forbid globals or restrict them to a safe subset.

Here is an example of an unpickler allowing only few safe classes from the :mod:`builtins` module to be loaded:

```
import builtins
import io
import pickle

safe_builtins = {
    'range',
    'complex',
    'set',
    'frozenset',
    'slice',
}

class RestrictedUnpickler(pickle.Unpickler):

    def find_class(self, module, name):
        # Only allow safe classes from builtins.
        if module == "builtins" and name in safe_builtins:
            return getattr(builtins, name)
        # Forbid everything else.
        raise pickle.UnpicklingError("global '%s.%s' is forbidden" %
                                     (module, name))

def restricted_loads(s):
    """Helper function analogous to pickle.loads()."""
    return RestrictedUnpickler(io.BytesIO(s)).load()
```

A sample usage of our unpickler working has intended:

```
>>> restricted_loads(pickle.dumps([1, 2, range(15)]))
[1, 2, range(0, 15)]
```

```
>>> restricted_loads(b"cos\nsystem\n(S'echo hello world'\ntR.")
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'os.system' is forbidden
>>> restricted_loads(b'cbuiltins\neval\n'
...                  b'(S\'getattr(__import__("os"), "system")'
...                  b'("echo hello world")\'\ntR.')
Traceback (most recent call last):
  ...
pickle.UnpicklingError: global 'builtins.eval' is forbidden
```

As our examples shows, you have to be careful with what you allow to be unpickled. Therefore if security is a concern, you may want to consider alternatives such as the marshalling API in :mod:`xmlrpc.client` or third-party solutions.

Performance

Recent versions of the pickle protocol (from protocol 2 and upwards) feature efficient binary encodings for several common features and built-in types. Also, the :mod:`pickle` module has a transparent optimizer written in C.

Examples

For the simplest code, use the :func:`dump` and :func:`load` functions.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3, 4+6j],
    'b': ("character string", b"byte string"),
    'c': {None, True, False}
}

with open('data.pickle', 'wb') as f:
    # Pickle the 'data' dictionary using the highest protocol available.
    pickle.dump(data, f, pickle.HIGHEST_PROTOCOL)
```

The following example reads the resulting pickled data.

```
import pickle

with open('data.pickle', 'rb') as f:
    # The protocol version used is detected automatically, so we do not
    # have to specify it.
    data = pickle.load(f)
```

```
.. seealso::

   Module :mod:`copyreg`
      Pickle interface constructor registration for extension types.

   Module :mod:`pickletools`
      Tools for working with and analyzing pickled data.

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

   Module :mod:`copy`
      Shallow and deep object copying.

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

Footnotes

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

[2] This is why :keyword:`lambda` functions cannot be pickled: all :keyword:`!lambda` functions share the same name: `<lambda>`.

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

[4] The :mod:`copy` module uses this protocol for shallow and deep copying operations.

[5] The limitation on alphanumeric characters is due to the fact the persistent IDs, in protocol 0, are delimited by the newline character. Therefore if any kind of newline characters occurs in persistent IDs, the resulting pickle will become unreadable.