pickle — Serialización de objetos Python

Código fuente: Lib/pickle.py

El modulo `pickle` implementa protocolos binarios para serializar y deserializar una estructura de objetos Python. *«Pickling»* es el proceso mediante el cual una jerarquía de objetos de Python se convierte en una secuencia de bytes, y el *«unpickling»* es la operación inversa, mediante la cual una secuencia de bytes de un archivo binario (binary file) ó un objeto tipo binario (bytes-like object) es convertido nuevamente en una jerarquía de objetos. *Pickling* (y *unpickling*) son alternativamente conocidos como «serialización», «ensamblaje,» [1] o «aplanamiento»; sin embargo, para evitar confusiones, los términos utilizados aquí son «pickling» y «unpickling».

Advertencia: El modulo `pickle` **no es seguro**. Solo deserialize con *pickle* los datos en los que confía.

Es posible construir datos maliciosos con *pickle* que **ejecuten código arbitrario durante el proceso de `unpickling`**. Nunca deserialize datos con *pickle* que podrían haber venido de una fuente no confiable, o que podrían haber sido manipulados.

Considere firmar los datos con `hmac` si necesita asegurarse de que no hayan sido alterados.

Los formatos de serialización más seguros como `json` pueden ser más apropiados si está procesando datos no confiables. Ver Comparación con json.

Relación con otros módulos de Python

Comparación con `marshal`

Python tiene un módulo de serialización más primitivo llamado `marshal`, pero en general `pickle` debería ser siempre la forma preferida de serializar objetos de Python. `marshal` existe principalmente para soportar archivos Python `.pyc`.

El modulo `pickle` difiere de `marshal` en varias formas significativas:

- El modulo `pickle` realiza un seguimiento de los objetos que ya ha serializado, para que las referencias posteriores al mismo objeto no se serializen nuevamente. `marshal` no hace esto.

  Esto tiene implicaciones tanto para los objetos recursivos como para compartir objetos. Los objetos recursivos son objetos que contienen referencias a sí mismos. *Marshal* no los maneja y, de hecho, intentar agrupar objetos recursivos bloqueará su intérprete de Python. El intercambio de objetos ocurre cuando hay múltiples referencias al mismo objeto en diferentes lugares de la jerarquía de objetos que se serializan. `pickle` almacena dichos objetos solo una vez y garantiza que todas las demás referencias apunten a la copia maestra. Los objetos compartidos permanecen compartidos, lo cual puede ser muy importante para los objetos mutables.

- `marshal` no se puede usar para serializar clases definidas por el usuario y sus instancias. `pickle` puede guardar y restaurar instancias de clase de forma transparente, sin embargo, la definición de clase debe ser importable y vivir en el mismo módulo que cuando se almacenó el objeto.

- No se garantiza que el formato de serialización `marshal` sea portable a través de todas las versiones de Python. Debido a que su trabajo principal es dar soporte a archivos `.pyc`, los implementadores de Python se reservan el derecho de cambiar el formato de serialización de formas no compatibles con versiones anteriores si surge la necesidad. El formato de serialización `pickle` está garantizado para ser compatible con versiones anteriores de Python siempre que se elija un protocolo de *pickle* compatible y el serializado y deserializado de código con *pickle* se encargue de lidiar con las diferencias de tipos entre Python 2 y Python 3 si sus datos están cruzando ese limite único entre las versiones del lenguaje.

Comparación con `json`

Existen diferencias fundamentales entre los protocolos de *pickle* y JSON (acrónimo de JavaScript Object Notation, «notación de objeto de JavaScript»):

- JSON es un formato de serialización de texto (genera texto unicode, aunque la mayoría de las veces se codifica a `utf-8`), mientras que *pickle* es un formato de serialización binario;
- JSON es legible por humanos, mientras que *pickle* no lo es;
- JSON es interoperable y ampliamente utilizado fuera del ecosistema de Python, mientras que *pickle* es específico de Python;
- JSON, por defecto, solo puede representar un subconjunto de los tipos integrados de Python, y no clases personalizadas; *pickle* puede representar un número extremadamente grande de tipos de Python (muchos de ellos automáticamente, mediante el uso inteligente de la introspección de objetos en Python; los casos complejos se pueden abordar implementando API de objetos específicos, specific object APIs);
- A diferencia de *pickle*, deserializar JSON no confiable no crea en sí mismo una vulnerabilidad de ejecución de código arbitraria.

Ver también: El modulo `json`: un módulo de la biblioteca estándar que permite la serialización y deserialización de JSON.

El formato de datos utilizado por `pickle` es específico de Python. Esto tiene la ventaja de que no hay restricciones impuestas por estándares externos como JSON o XDR (que no pueden representar el uso compartido de punteros); sin embargo, significa que los programas que no son de Python pueden no ser capaces de reconstruir objetos Python serialzados con *pickle*.

Por defecto, el formato de datos `pickle` utiliza una representación binaria relativamente compacta. Si necesita características de tamaño óptimas, puede eficientemente comprimir datos serializados con *pickle*.

El modulo `pickletools` contiene herramientas para analizar flujos de datos generados por `pickle`. El código fuente de `pickletools` tiene comentarios extensos sobre los códigos de operación utilizados por los protocolos de *pickle*.

Actualmente hay 6 protocolos diferentes que se pueden utilizar para serializar con *pickle*. Cuanto mayor sea el protocolo utilizado, más reciente será la versión de Python necesaria para leer el *pickle* producido.

- La versión 0 del protocolo es el protocolo original «legible para humanos» y es compatible con versiones anteriores de Python.
- La versión 1 del protocolo es un formato binario antiguo que también es compatible con versiones anteriores de 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 information about improvements brought by protocol 2.
- Se agregó la versión 3 del protocolo en Python 3.0. Tiene soporte explícito para objetos `bytes` y no puede ser deserializado con *pickle* por Python 2.x. Este era el protocolo predeterminado en Python 3.0–3.7.
- Se agregó la versión 4 del protocolo en Python 3.4. Agrega soporte para objetos muy grandes, *pickling* de mas tipos de objetos y algunas optimizaciones de formato de datos. Es el protocolo predeterminado que comienza con Python 3.8. Consulte **PEP 3154** para obtener información sobre las mejoras aportadas por el protocolo 4.
- Se agregó la versión 5 del protocolo en Python 3.8. Agrega soporte para datos fuera de banda y aceleración para datos dentro de banda. Consulte **PEP 574** para obtener información sobre las mejoras aportadas por el protocolo 5.

Nota: La serialización es una noción más primitiva que la persistencia; aunque `pickle` lee y escribe objetos de archivo, no maneja el problema de nombrar objetos persistentes, ni el problema (aún más complicado) de acceso concurrente a objetos persistentes. El módulo `pickle` puede transformar un objeto complejo en una secuencia de bytes y puede transformar la secuencia de bytes en un objeto con la misma estructura interna. Quizás lo más obvio que hacer con estos flujos de bytes es escribirlos en un archivo, pero también es concebible enviarlos a través de una red o almacenarlos en una base de datos. El módulo `shelve` proporciona una interfaz simple para serializar y deserializar objetos con *pickle* en archivos de bases de datos de estilo DBM.

Interfaz del módulo

Para serializar una jerarquía de objetos, simplemente llame a la función `dumps()`. De manera similar, para deserializar un flujo de datos, llama a la función `loads()`. Sin embargo, si desea tener más control sobre la serialización y la deserialización, puede crear un objeto `Pickler` o `Unpickler`, respectivamente.

El módulo `pickle` proporciona las siguientes constantes:

`pickle`.**HIGHEST_PROTOCOL**

Un entero, la versión de protocolo (protocol version) más alta disponible. Este valor se puede pasar como un valor de *protocolo* a las funciones `dump()` y `dumps()` así como al constructor `Pickler`.

`pickle`.**DEFAULT_PROTOCOL**

Un entero, la versión de protocolo (protocol version) predeterminada utilizada para el serializado con *pickle*. Puede ser menor que `HIGHEST_PROTOCOL`. Actualmente, el protocolo predeterminado es 4, introducido por primera vez en Python 3.4 e incompatible con versiones anteriores.

Distinto en la versión 3.0: El protocolo predeterminado es 3.

Distinto en la versión 3.8: El protocolo predeterminado es 4.

El módulo `pickle` proporciona las siguientes funciones para que el proceso de *pickling* sea más conveniente:

`pickle`.**dump**(*obj*, *file*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

Escribe la representación *pickle* del objeto *obj* en el *archivo* abierto file object. Esto es equivalente a `Pickler(file, protocol).dump(obj)`.

Los argumentos *file*, *protocol*, *fix_imports* y *buffer_callback* tienen el mismo significado que en el constructor `Pickler`.

Distinto en la versión 3.8: Se agregó el argumento *buffer_callback*.

`pickle`.**dumps**(*obj*, *protocol=None*, ***, *fix_imports=True*, *buffer_callback=None*)

Retorna la representación *pickle* del objeto *obj* como un objeto `bytes`, en lugar de escribirlo en un archivo.

Los argumentos *protocol*, *fix_imports* y *buffer_callback* tienen el mismo significado que en el constructor `Pickler`.

Distinto en la versión 3.8: Se agregó el argumento *buffer_callback*.

pickle.**load**(*file*, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Lee la representación *pickle* de un objeto desde un *archivo* abierto file object y retorna la jerarquía de objetos reconstituidos especificada en el mismo. Esto es equivalente a `Unpickler(file).load()`.

La versión de protocolo del *pickle* se detecta automáticamente, por lo que no se necesita ningún argumento de protocolo. Los bytes más allá de la representación empaquetada son ignorados.

Los argumentos *file*, *fix_imports*, *encoding*, *errors*, *strict* y *buffers* tienen el mismo significado que en el constructor `Unpickler`.

Distinto en la versión 3.8: Se agregó el argumento *buffers*.

pickle.**loads**(*data*, */*, *, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Retorna la jerarquía de objetos reconstruida de la representación *pickle* *data* de un objeto. *data* debe ser un objeto tipo binario (bytes-like object).

La versión de protocolo del *pickle* se detecta automáticamente, por lo que no se necesita ningún argumento de protocolo. Los bytes más allá de la representación empaquetada son ignorados.

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

Distinto en la versión 3.8: Se agregó el argumento *buffers*.

El módulo `pickle` define tres excepciones:

exception pickle.**PickleError**

Clase base común para las otras excepciones de *pickling*. Hereda de `Exception`.

exception pickle.**PicklingError**

Error generado cuando `Pickler` encuentra un objeto que no se puede serializar con *pickle* . Hereda de `PickleError`.

Consulte ¿Qué se puede serializar (pickled) y deserializar (unpickled) con pickle? para aprender qué tipos de objetos se pueden serializar con *pickle*.

exception pickle.**UnpicklingError**

Se produce un error cuando hay un problema al deserializar un objeto con *pickle*, por ejemplo como una corrupción de datos o una violación de seguridad. Hereda de `PickleError`.

Tenga en cuenta que también se pueden generar otras excepciones durante la deserializacion con *pickle*, incluyendo (pero no necesariamente limitado a) *AttributeError*, *EOFError*, *ImportError*, e *IndexError*.

El módulo `pickle` exporta tres clases, `Pickler`, `Unpickler` y `PickleBuffer`:

class pickle.**Pickler**(*file*, *protocol=None*, *, *fix_imports=True*, *buffer_callback=None*)

Esto toma un archivo binario para escribir un flujo de datos de *pickle*.

El argumento opcional *protocol* , un entero, le dice al *pickler* que use el protocolo dado; los protocolos admitidos son 0 para `HIGHEST_PROTOCOL`. Si no se especifica, el valor predeterminado es `DEFAULT_PROTOCOL`. Si se especifica un número negativo, `HIGHEST_PROTOCOL` es seleccionado.

El argumento *file* debe tener un método *write()* que acepte un argumento de bytes individuales. Por lo tanto, puede ser un archivo en disco abierto para escritura binaria, una instancia `io.BytesIO` , o cualquier otro objeto personalizado que cumpla con esta interfaz.

Si *fix_imports* es verdadero y *protocol* es menor que 3, *pickle* intentará asignar los nuevos nombres de Python 3 a los nombres de módulos antiguos utilizados en Python 2, de modo que la secuencia de datos de *pickle* sea legible con Python 2.

Si *buffer_callback* es None (el valor predeterminado), las vistas de búfer se serializan en *file* como parte de la secuencia de *pickle*.

Si *buffer_callback* no es None, entonces se puede llamar cualquier número de veces con una vista de búfer. Si la *callback* retorna un valor falso (como None), el búfer dado está fuera de banda (out-of-band); de lo contrario, el búfer se serializa en banda, es decir, dentro del flujo de *pickle*.

Es un error si *buffer_callback* no es None y *protocol* es None o menor que 5.

Distinto en la versión 3.8: Se agregó el argumento *buffer_callback*.

dump(*obj*)

Escribe la representación serializada con *pickle* del objeto *obj* en el objeto archivo abierto dado en el constructor.

No hacer nada por defecto. Esto existe para que una subclase pueda sobreescribirlo.

Si `persistent_id()` retorna `None`, *obj* es serializado con *pickle* como siempre. Cualquier otro valor hace que `Pickler` emita el valor retornado como un ID persistente para *obj*. El significado de este ID persistente debe definirse por `Unpickler.persistent_load()`. Tenga en cuenta que el valor retornado por `persistent_id()` no puede tener una ID persistente.

Ver Persistencia de objetos externos para detalles y ejemplos de uso.

`dispatch_table`

La tabla de envío de un objeto *Pickler* es un registro de *funciones de reducción* del tipo que se puede declarar usando `copyreg.pickle()`. Es un mapeo cuyas claves son clases y cuyos valores son funciones de reducción. Una función de reducción toma un solo argumento de la clase asociada y debe ajustarse a la misma interfaz que un método `__reduce__()`.

Por defecto, un objeto de *pickle* no tendrá un atributo `dispatch_table`, y en su lugar utilizará la tabla de despacho global administrada por el módulo `copyreg`. Sin embargo, para personalizar el *pickling* para un objeto de *pickle* específico, se puede establecer el atributo `dispatch_table` en un objeto tipo dict. Alternativamente, si una subclase de `Pickler` tiene un atributo `dispatch_table` esto se usará como la tabla de despacho predeterminada para instancias de esa clase.

Ver Tablas de despacho para ejemplos de uso.

Nuevo en la versión 3.3.

`reducer_override`(*obj*)

Reductor especial que se puede definir en subclases de `Pickler`. Este método tiene prioridad sobre cualquier reductor en `dispatch_table`. Debe cumplir con la misma interfaz que un método `__reduce__()`, y opcionalmente puede retornar `NotImplemented` para recurrir a reductores registrados en `dispatch_table` el objeto *pickle* `obj`.

Para un ejemplo detallado, ver Reducción personalizada para tipos, funciones y otros objetos.

Nuevo en la versión 3.8.

`fast`

Obsoleto. Habilite el modo rápido si se establece en un valor verdadero. El modo rápido deshabilita el uso de memo, por lo tanto, acelera el proceso de *pickling* al no generar códigos de operación PUT superfluos. No debe usarse con objetos autorreferenciales; de lo contrario, la clase `Pickler` se repetirá infinitamente.

Use `pickletools.optimize()` si necesita *pickles* más compactos.

class `pickle.`**`Unpickler`**(*file*, ***, *fix_imports=True*, *encoding='ASCII'*, *errors='strict'*, *buffers=None*)

Esto toma un archivo binario para leer un flujo de datos de *pickle*.

La versión de protocolo de *pickle* se detecta automáticamente, por lo que no se necesita ningún argumento de protocolo.

El argumento *file* debe tener tres métodos, un método read() que toma un argumento entero, un método *readinto()* que toma un argumento búfer y un método *readline()* que no requiere argumentos, como en la interfaz `io.BufferedIOBase`. Por lo tanto *file* puede ser un archivo en disco abierto para lectura binaria, un objeto `io.BytesIO`, o cualquier otro objeto personalizado que cumpla con esta interfaz.

Los argumentos opcionales *fix_imports*, *encoding* and *errors* se utilizan para controlar el soporte de compatibilidad para el flujo de *pickle* generado por Python 2. Si *fix_imports* es verdadero, *pickle* intentará asignar los nombres antiguos de Python 2 a los nuevos nombres utilizados en Python 3. Tanto *encoding* como *errors* le indican a *pickle* cómo decodificar instancias de cadenas de 8 bits seleccionadas por Python 2; estos son predeterminados a “ASCII” y “strict”, respectivamente. *encoding* puede ser “bytes” para leer estas instancias de cadena de 8 bits como objetos de bytes. Se requiere el uso de `encoding='latin1'` para realizar el *unpickling* de arreglos de NumPy e instancias de `datetime`, `date` y `time` serializados con *pickle* por Python 2.

Si *buffers* es None (el valor predeterminado), todos los datos necesarios para la deserialización deben estar contenidos en el flujo de *pickle*. Esto significa que el argumento *buffer_callback* era None cuando se instanciaba una clase `Pickler` (o cuando se llamaba a `dump()` o `dumps()`).

Si *buffers* no es None, debería ser un iterable de objetos habilitados para almacenamiento intermedio que se consumen cada vez que el flujo de *pickle* hace referencia a una vista de buffer fuera de banda (out-of-band). Tales buffers se han dado para el *buffer_callback* de un objeto *Pickler*.

Distinto en la versión 3.8: Se agregó el argumento *buffers*.

`load`()

Lee la representación serializada con *pickle* de un objeto desde el objeto de archivo abierto dado en el constructor, y retorne la

persistent_load(*pid*)

Lanza un `UnpicklingError` de forma predeterminada.

Si se define, `persistent_load()` debería retornar el objeto especificado por el ID persistente *pid*. Si se encuentra un ID persistente no válido, se debe lanzar un `UnpicklingError`.

Ver Persistencia de objetos externos para detalles y ejemplos de uso.

find_class(*module*, *name*)

Importa *module* si es necesario y retorna el objeto llamado *name* desde el, donde los argumentos *module* y *name* son objetos de `str`. Tenga en cuenta que, a diferencia de lo que sugiere su nombre, `find_class()` también se usa para buscar funciones.

Las subclases pueden sobreescribir esto para obtener control sobre qué tipo de objetos y cómo se pueden cargar, reduciendo potencialmente los riesgos de seguridad. Consulte Restricción de Globals para obtener más detalles.

Lanza un auditing event `pickle.find_class` con argumentos `module`, `name`.

class `pickle`.**PickleBuffer**(*buffer*)

Un envoltorio (*wrapper*) para un búfer que representa datos serializables con *pickle* (*picklable data*). *buffer* debe ser un objeto que proporciona un búfer (buffer-providing), como objeto tipo binario (bytes-like object) o un arreglo N-dimensional.

`PickleBuffer` es en sí mismo un proveedor de búfer, por lo que es posible pasarlo a otras API que esperan un objeto que provea un búfer, como `memoryview`.

Los objetos `PickleBuffer` solo se pueden serializar usando el protocolo *pickle* 5 o superior. Son elegibles para serialización fuera de banda (out-of-band serialization).

Nuevo en la versión 3.8.

raw()

Retorna un `memoryview` del área de memoria subyacente a este búfer. El objeto retornado es una vista de memoria unidimensional, C-contigua con formato `B` (bytes sin firmar). `BufferError` es lanzado si el búfer no es contiguo a C ni a Fortran.

release()

Libera el búfer subyacente expuesto por el objeto PickleBuffer.

¿Qué se puede serializar (pickled) y deserializar (unpickled) con *pickle*?

Los siguientes tipos se pueden serializar con *pickle* (pickled):

- `None`, `True`, and `False`;
- integers, floating-point numbers, complex numbers;
- strings, bytes, bytearrays;
- tuples, lists, sets, and dictionaries containing only picklable objects;
- functions (built-in and user-defined) accessible from the top level of a module (using `def`, not `lambda`);
- classes accessible from the top level of a module;
- instancias de tales clases cuyo `__dict__` o el resultado de llamar a `__getstate__()` es serializable con *pickle* (picklable) (consulte la sección Pickling de Instancias de clases para obtener más detalles).

Los intentos de serializar objetos no serializables con *pickle* lanzaran la excepción `PicklingError`; cuando esto sucede, es posible que ya se haya escrito una cantidad no especificada de bytes en el archivo subyacente. Intentar serializar con *pickle* una estructura de datos altamente recursiva puede exceder la profundidad máxima de recursividad, en este caso se lanzará un `RecursionError`. Puede aumentar cuidadosamente este límite con `sys.setrecursionlimit()`.

Note that functions (built-in and user-defined) are pickled by fully qualified name, not by value. [2] This means that only the function name is pickled, along with the name of the containing module and classes. 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 fully qualified name, 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)
```

De manera similar, cuando las instancias de clases son serializadas con *pickle*, el código y los datos de la clase no son serializadas junto con ella. Solo los datos de la instancia son serializados con *pickle* (*pickled*). Esto se hace a propósito, por lo que puede corregir errores en una clase o agregar métodos a la clase y aún cargar objetos que fueron creados con una versión anterior de la clase. Si planea tener objetos de larga duración que verán muchas versiones de una clase, puede valer la pena poner un número de versión en los objetos para que las conversiones adecuadas se puedan realizar mediante el método `__setstate__()`.

Pickling de Instancias de clases

En esta sección, describimos los mecanismos generales disponibles para que usted defina, personalice y controle cómo se serializan y deserializan con *Pickle* las instancias de clase.

En la mayoría de los casos, no se necesita código adicional para hacer que las instancias sean *picklable* (serializables con *pickle*). Por defecto, *pickle* recuperará la clase y los atributos de una instancia a través de la introspección. Cuando una instancia de clase es deserializada con *pickle* (*unpickled*), su método `__init__()` generalmente *no* se invoca. El comportamiento predeterminado es que primero crea una instancia no inicializada y luego restaura los atributos guardados. El siguiente código muestra una implementación de este comportamiento:

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

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

Las clases pueden alterar el comportamiento predeterminado proporcionando uno o varios métodos especiales:

`object.`**`__getnewargs_ex__`**`()`

En los protocolos 2 y más recientes, las clases que implementan el método `__getnewargs_ex__()` pueden dictar los valores pasados al método `__new__()` al hacer`unpickling`. El método debe retornar un par `(args, kwargs)` donde *args* es una tupla de argumentos posicionales y *kwargs* un diccionario de argumentos con nombre para construir el objeto. Estos se pasarán al método `__new__()` al hacer *unpickling*.

Debes implementar este método si el método `__new__()` de tu clase requiere argumentos de solo palabras clave. De lo contrario, se recomienda para la compatibilidad implementar `__getnewargs__()`.

Distinto en la versión 3.6: `__getnewargs_ex__()` ahora se usa en los protocolos 2 y 3.

`object.`**`__getnewargs__`**`()`

Este método tiene un propósito similar a `__getnewargs_ex__()`, pero solo admite argumentos posicionales. Debe retornar una tupla de argumentos `args` que se pasarán al método `__new__()` al hacer *unpickling*.

`__getnewargs__()` no se llamará si `__getnewargs_ex__()` está definido.

Distinto en la versión 3.6: Antes de Python 3.6, se llamaba a, `__getnewargs__()` en lugar de `__getnewargs_ex__()` en los protocolos 2 y 3.

`object.`**`__getstate__`**`()`

Las clases pueden influir aún más en cómo sus instancias se serializan con *pickle*; si la clase define el método `__getstate__()`, este es llamado y el objeto retornado se selecciona como contenido de la instancia, en lugar del contenido del diccionario de la instancia. Si el método `__getstate__()` está ausente, el `__dict__` de la instancia se conserva como de costumbre.

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

Al hacer *unpickling*, si la clase define `__setstate__()`, este es llamado con el estado *unpickled* (no serializado con *pickle*). En ese caso, no es necesario que el objeto de estado sea un diccionario. De lo contrario, el estado *pickled* (*pickled state*) debe ser un diccionario y sus elementos se asignan al diccionario de la nueva instancia.

> **Nota:** Si `__getstate__()` retorna un valor falso, el método `__setstate__()` no se llamará al hacer *unpickling*.

Consulte la sección Manejo de objetos con estado para obtener más información sobre cómo utilizar los métodos `__getstate__()` y `__setstate__()`.

> **Nota:** Al momento de hacer *unpickling*, algunos métodos como `__getattr__()`, `__getattribute__()`, o `__setattr__()` pueden invocarse sobre la instancia. En caso de que esos métodos dependan de que algún invariante interno sea verdadero, el tipo debería implementar `__new__()` para establecer tal invariante, ya que `__init__()` no se llama cuando se hace *unpickling* de una instancia.

Como veremos, *pickle* no utiliza directamente los métodos descritos anteriormente. De hecho, estos métodos son parte del protocolo de

necesarios para hacer el *pickling* y la copia de objetos. [4]

Aunque es poderoso, implementar `__reduce__()` directamente en sus clases es propenso a errores. Por esta razón, los diseñadores de clases deben usar la interfaz de alto nivel (es decir, `__getnewargs_ex__()`, `__getstate__()` y `__setstate__()`) siempre que sea posible. Sin embargo, mostraremos casos en los que usar `__reduce__()` es la única opción o conduce a un *pickling* más eficiente o ambos.

`object.__reduce__()`

La interfaz se define actualmente de la siguiente manera. El método `__reduce__()` no toma ningún argumento y retornará una cadena o preferiblemente una tupla (el objeto retornado a menudo se denomina «valor reducido»).

Si se retorna una cadena, la cadena debe interpretarse como el nombre de una variable global. Debe ser el nombre local del objeto relativo a su módulo; el módulo *pickle* busca en el espacio de nombres del módulo para determinar el módulo del objeto. Este comportamiento suele ser útil para singletons.

Cuando se retorna una tupla, debe tener entre dos y seis elementos. Los elementos opcionales se pueden omitir o se puede proporcionar `None` como su valor. La semántica de cada elemento está en orden:

- Un objeto invocable que se llamará para crear la versión inicial del objeto.
- Una tupla de argumentos para el objeto invocable. Se debe proporcionar una tupla vacía si el invocable no acepta ningún argumento.
- Opcionalmente, el estado del objeto, que se pasará al método `__setstate__()` del objeto como se describió anteriormente. Si el objeto no tiene dicho método, el valor debe ser un diccionario y se agregará al atributo `__dict__` del objeto.
- Opcionalmente, un iterador (y no una secuencia) produce elementos sucesivos. Estos elementos se agregarán al objeto usando `obj.append(item)` o, por lotes, usando `obj.extend(list_of_items)`. Esto se usa principalmente para subclases de lista, pero puede ser usado por otras clases siempre que tengan los métodos `append()` y `extend()` con la firma apropiada. (El uso de `append()` o `extend()` depende de la versión del protocolo *pickle* que se use, así como de la cantidad de elementos que se agregarán, por lo que ambos deben ser soportados.)
- Opcionalmente, un iterador (no una secuencia) que produce pares clave-valor sucesivos. Estos elementos se almacenarán en el objeto usando `obj[key] = value`. Esto se usa principalmente para subclases de diccionario, pero otras clases pueden usarlo siempre que implementen `__setitem__()`.
- Opcionalmente, un invocable con una firma `(obj, state)`. Este invocable permite al usuario controlar programáticamente el comportamiento de actualización de estado de un objeto específico, en lugar de usar el método estático de `obj __setstate__()`. Si no es `None`, este invocable tendrá prioridad sobre `obj`'s `__setstate__()`.

  Nuevo en la versión 3.8: Se agregó el sexto elemento opcional de tupla `(obj, state)`.

`object.__reduce_ex__(protocol)`

Alternativamente, se puede definir un método `__reduce_ex__()`. La única diferencia es que este método debe tomar un único argumento entero, la versión del protocolo. Cuando esté definido, *pickle* lo preferirá en lugar del método `__reduce__()`. Además, `__reduce__()` se convierte automáticamente en sinónimo de la versión extendida. El uso principal de este método es proporcionar valores reducidos compatibles con versiones anteriores para versiones anteriores de Python.

Persistencia de objetos externos

Para el beneficio de la persistencia del objeto, el módulo `pickle` admite la noción de una referencia a un objeto fuera del flujo de datos serializados con *pickle*. Dichos objetos son referenciados por un ID persistente, que debe ser una cadena de caracteres alfanuméricos (para el protocolo 0) [5] o simplemente un objeto arbitrario (para cualquier protocolo más nuevo).

La resolución de tales ID persistentes no está definida por el módulo `pickle`; delegará esta resolución a los métodos definidos por el usuario en el *pickler* y el *unpickler*, `persistent_id()` y `persistent_load()` respectivamente.

Para seleccionar objetos que tienen una ID persistente externo, el *pickler* debe tener un método personalizado `persistent_id()` que toma un objeto como argumento y retorna `None` o el ID persistente para ese objeto. Cuando se retorna `None`, el *pickler* simplemente serializará el objeto de forma normal. Cuando se retorna una cadena de identificación persistente, el *pickler* serializará ese objeto, junto con un marcador para que el *unpickler* lo reconozca como una identificación persistente.

Para hacer el *unpickling* objetos externos, el *unpickler* debe tener un método personalizado `persistent_load()` que toma un objeto de identificación persistente y retorna el objeto referenciado.

Aquí hay un ejemplo completo que presenta cómo se puede usar la identificación persistente para hacer el *pickling* objetos externos por referencia.

```
# Simple example presenting how persistent ID can be used to pickle
# external objects by reference.
```

```
import sqlite3
from collections import namedtuple

# Simple class representing a record in our database.
MemoRecord = namedtuple("MemoRecord", "key, task")

class DBPickler(pickle.Pickler):

    def persistent_id(self, obj):
        # Instead of pickling MemoRecord as a regular class instance, we emit a
        # persistent ID.
        if isinstance(obj, MemoRecord):
            # Here, our persistent ID is simply a tuple, containing a tag and a
            # key, which refers to a specific record in the database.
            return ("MemoRecord", obj.key)
        else:
            # If obj does not have a persistent ID, return None. This means obj
            # needs to be pickled as usual.
            return None


class DBUnpickler(pickle.Unpickler):

    def __init__(self, file, connection):
        super().__init__(file)
        self.connection = connection

    def persistent_load(self, pid):
        # This method is invoked whenever a persistent ID is encountered.
        # Here, pid is the tuple returned by DBPickler.
        cursor = self.connection.cursor()
        type_tag, key_id = pid
        if type_tag == "MemoRecord":
            # Fetch the referenced record from the database and return it.
            cursor.execute("SELECT * FROM memos WHERE key=?", (str(key_id),))
            key, task = cursor.fetchone()
            return MemoRecord(key, task)
        else:
            # Always raises an error if you cannot return the correct object.
            # Otherwise, the unpickler will think None is the object referenced
            # by the persistent ID.
            raise pickle.UnpicklingError("unsupported persistent object")


def main():
    import io
    import pprint

    # Initialize and populate our database.
    conn = sqlite3.connect(":memory:")
    cursor = conn.cursor()
    cursor.execute("CREATE TABLE memos(key INTEGER PRIMARY KEY, task TEXT)")
    tasks = (
        'give food to fish',
        'prepare group meeting',
        'fight with a zebra',
        )
    for task in tasks:
        cursor.execute("INSERT INTO memos VALUES(NULL, ?)", (task,))

    # Fetch the records to be pickled.
    cursor.execute("SELECT * FROM memos")
    memos = [MemoRecord(key, task) for key, task in cursor]
    # Save the records using our custom DBPickler.
    file = io.BytesIO()
    DBPickler(file).dump(memos)

    print("Pickled records:")
    pprint.pprint(memos)

    # Update a record, just for good measure.
    cursor.execute("UPDATE memos SET task='learn italian' WHERE key=1")

    # Load the records from the pickle data stream.
    file.seek(0)
    memos = DBUnpickler(file, conn).load()

    print("Unpickled records:")
```

```
if __name__ == '__main__':
    main()
```

Tablas de despacho

Si se desea personalizar el *pickling* de algunas clases sin alterar ningún otro código que dependa del *pickling*, se puede crear un *pickler* con una tabla de despacho privada.

La tabla de despacho global administrada por el módulo `copyreg` está disponible como `copyreg.dispatch_table`. Por lo tanto, se puede optar por utilizar una copia modificada de `copyreg.dispatch_table` como tabla de envío privada.

Por ejemplo

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

crea una instancia de `pickle.Pickler` con una tabla de despacho privada que maneja la clase `AlgunaClase` especialmente. Alternativamente, el código

```
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 private dispatch table. On the other hand, the code

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

modifies the global dispatch table shared by all users of the `copyreg` module.

Manejo de objetos con estado

Aquí hay un ejemplo que muestra cómo modificar el comportamiento del *pickling* de una clase. La clase `TextReader` abre un archivo de texto y retorna el número de línea y el contenido de la línea cada vez que se llama a su método `readline()` . Si se selecciona una instancia de `TextReader` se guardan todos los atributos *excepto* el miembro del objeto de archivo. Cuando se hace el *unpickling* de la instancia, el archivo se vuelve a abrir y la lectura se reanuda desde la última ubicación. Los métodos `__setstate__()` y `__getstate__()` se utilizan para implementar este comportamiento.

```
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)
```

```
        # 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
```

Un ejemplo de uso podría ser algo como esto:

```
>>> 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!'
```

Reducción personalizada para tipos, funciones y otros objetos

Nuevo en la versión 3.8.

A veces, `dispatch_table` puede no ser lo suficientemente flexible. En particular, es posible que deseemos personalizar el *pickling* en función de otro criterio que no sea el tipo de objeto, o es posible que deseemos personalizar el *pickling* de funciones y clases.

Para esos casos, es posible crear una subclase de la clase `Pickler` e implementar el método `reducer_override()`. Este método puede retornar una tupla de reducción arbitraria (ver `__reduce__()`). Alternativamente, puede retornar `NotImplemented` para volver al comportamiento tradicional.

Si se definen tanto `dispatch_table` como `reducer_override()`, entonces `reducer_override()` tiene prioridad.

Nota: Por motivos de rendimiento, no se puede llamar a `reducer_override()` para los siguientes objetos: `None`, `True`, `False`, e instancias exactas de `int`, `float`, `bytes`, `str`, `dict`, `set`, `frozenset`, `list` y `tuple`.

Aquí hay un ejemplo simple donde permitimos el *pickling* y reconstruir una clase dada 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
```

Búferes fuera de banda

Nuevo en la versión 3.8.

En algunos contextos, el módulo `pickle` se usa para transferir cantidades masivas de datos. Por lo tanto, puede ser importante minimizar el número de copias de memoria para preservar el rendimiento y el consumo de recursos. Sin embargo, el funcionamiento normal del módulo `pickle`, ya que transforma una estructura gráfica de objetos en un flujo secuencial de bytes, implica intrínsecamente copiar datos hacia y desde el flujo `pickle`.

Esta restricción puede evitarse si tanto el *proveedor* (la implementación de los tipos de objeto a transferir) como el *consumidor* (a implementación del sistema de comunicaciones) admiten las facilidades de transferencia fuera de banda proporcionadas por el protocolo *pickle* 5 y mayor.

API de proveedor

Los objetos de datos grandes que se van a serializar con *pickle* deben implementar un método `__reduce_ex__()` especializado para el protocolo 5 y superior, que retorna una instancia de `PickleBuffer` (en lugar de, por ejemplo, un objeto `bytes` object) para cualquier datos.

Un objeto `PickleBuffer` *indica* que el búfer subyacente es elegible para la transferencia de datos fuera de banda. Estos objetos siguen siendo compatibles con el uso normal del módulo `pickle` . Sin embargo, los consumidores también pueden optar por decirle a `pickle` que manejarán esos búferes por sí mismos.

API de consumidor

Un sistema de comunicaciones puede permitir el manejo personalizado de los objetos `PickleBuffer` generados al serializar un gráfico de objetos.

En el lado del envío, necesita pasar un argumento *buffer_callback* a `Pickler` (o a las funciones `dump()` o `dumps()`), que se llamará con cada `PickleBuffer` generado al hacer *pickling* del gráfico del objeto. Los búferes acumulados por *buffer_callback* no verán sus datos copiados en el flujo de *pickle*, solo se insertará un marcador barato.

En el lado receptor, necesita pasar un argumento *buffers* a `Unpickler` (o a las funciones `load()` o `loads()`), que es un iterable de los búferes que fueron pasado a *buffer_callback*. Ese iterable debería producir búferes en el mismo orden en que se pasaron a *buffer_callback*. Esos búferes proporcionarán los datos esperados por los reconstructores de los objetos cuyo *pickling* produjo los objetos originales `PickleBuffer`.

Entre el lado de envío y el lado de recepción, el sistema de comunicaciones es libre de implementar su propio mecanismo de transferencia para memorias intermedias fuera de banda. Las posibles optimizaciones incluyen el uso de memoria compartida o compresión dependiente del tipo de datos.

Ejemplo

Aquí hay un ejemplo trivial donde implementamos una subclase `bytearray` capaz de participar en el *pickling* de un búfer fuera de banda:

```
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)
```

El reconstructor (el método de clase `_reconstruct`) retorna el objeto que proporciona el búfer si tiene el tipo correcto. Esta es una manera fácil de simular el comportamiento de copia cero en este ejemplo de juguete.

En el lado del consumidor, podemos serializar con *pickle* esos objetos de la forma habitual, que cuando no se serializan nos dará una copia del objeto original:

```
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
```

Pero si pasamos un *buffer_callback* y luego retornamos los búferes acumulados al anular la serialización, podemos recuperar el objeto original:

```
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
```

Este ejemplo está limitado por el hecho de que `bytearray` asigna su propia memoria: no puedes crear una instancia de `bytearray` que esté respaldada por la memoria de otro objeto. Sin embargo, los tipos de datos de terceros, como las matrices NumPy no tienen esta limitación y permiten el uso de *pickling* de copia cero (o realizar la menor cantidad de copias posible) cuando se transfieren entre procesos o sistemas distintos.

Ver también: PEP 574 – Protocolo Pickle 5 con datos fuera de banda

Restricción de Globals

De forma predeterminada, el *unpickling* importará cualquier clase o función que encuentre en los datos de *pickle*. Para muchas aplicaciones, este comportamiento es inaceptable, ya que permite al *unpickler* importar e invocar código arbitrario. Solo considere lo que hace este flujo de datos de *pickle* hechos a mano cuando se carga:

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

En este ejemplo, el *unpickler* importa la función `os.system()` y luego aplica el argumento de cadena *«echo hello world»*. Aunque este ejemplo es inofensivo, no es difícil imaginar uno que pueda dañar su sistema.

Por esta razón, es posible que desee controlar lo que se deserializa con *pickle* personalizando `Unpickler.find_class()`. A diferencia de lo que sugiere su nombre, `Unpickler.find_class()` se llama siempre que se solicita un global (es decir, una clase o una función). Por lo tanto, es posible prohibir completamente los globales o restringirlos a un subconjunto seguro.

Aquí hay un ejemplo de un *unpickler* que permite cargar solo unas pocas clases seguras del módulo `builtins`:

```
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 as 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
```

Como muestran nuestros ejemplos, debes tener cuidado con lo que permites que se deserialize con *pickle*. Por lo tanto, si la seguridad es un problema, puede considerar alternativas como la API de *marshalling* en `xmlrpc.client` o soluciones de terceros.

Performance

Las versiones recientes del protocolo *pickle* (desde el protocolo 2 en adelante) cuentan con codificaciones binarias eficientes para varias características comunes y tipos integrados. Además, el módulo `pickle` tiene un optimizador transparente escrito en C.

Ejemplos

Para obtener el código más simple, use las funciones `dump()` y `load()`.

```
import pickle

# An arbitrary collection of objects supported by pickle.
data = {
    'a': [1, 2.0, 3+4j],
    '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)
```

El siguiente ejemplo lee los datos serializados con *pickle* resultantes.

```
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)
```

Ver también:

Módulo `copyreg`
: Registro de constructor de interfaz *Pickle* para tipos de extensión.

Módulo `pickletools`
: Herramientas para trabajar y analizar datos serializados con *pickle*.

Módulo `shelve`
: Bases de datos indexadas de objetos; usa `pickle`.

Module `copy`
: Copia de objetos superficial y profunda.

Módulo `marshal`
: Serialización de alto rendimiento de tipos integrados.

Notas al pie

[1] No confunda esto con el módulo `marshal`

[2] Esta es la razón por la que las funciones `lambda` no se pueden serializar con *pickle*: todas las funciones `lambda` comparten el mismo nombre: `<lambda>`.

[3] La excepción generada probablemente será un `ImportError` o un `AttributeError` pero podría ser otra cosa.

[4] El módulo `copy` utiliza este protocolo para operaciones de copia superficial y profunda.

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