persistent Documentation

Release 4.2.2

ZODB Developers <zope-dev@zope.org>

Sep 16, 2022

Contents

This package contains a generic persistence implementation for Python. It forms the core protocol for making objects interact "transparently" with a database such as the ZODB.

Contents:

CHAPTER 1

Using `persistent` in your application

1.1 Inheriting from `persistent.Persistent`

The basic mechanism for making your application's objects persistent is mix-in inheritance. Instances whose classes derive from *`persistent.Persistent`* are automatically capable of being created as *ghost* instances, being associated with a database connection (called the *jar*), and notifying the connection when they have been changed.

1.2 Relationship to a Data Manager and its Cache

Except immediately after their creation, persistent objects are normally associated with a *data manager* (also referred to as a *jar*). An object's data manager is stored in its `_p_jar` attribute. The data manager is responsible for loading and saving the state of the persistent object to some sort of backing store, including managing any interactions with transaction machinery.

Each data manager maintains an *object cache*, which keeps track of the currently loaded objects, as well as any objects they reference which have not yet been loaded: such an object is called a *ghost*. The cache is stored on the data manager in its `_cache` attribute.

A persistent object remains in the ghost state until the application attempts to access or mutate one of its attributes: at that point, the object requests that its data manager load its state. The persistent object also notifies the cache that it has been loaded, as well as on each subsequent attribute access. The cache keeps a "most-recently-used" list of its objects, and removes objects in least-recently-used order when it is asked to reduce its working set.

The examples below use a stub data manager class:

```
>>> from zope.interface import implementer
>>> from persistent.interfaces import IPersistentDataManager
>>> @implementer(IPersistentDataManager)
... class DM(object):
...     def __init__(self):
...         self.registered = 0
...     def register(self, ob):
```

(continues on next page)

(continued from previous page)

```
...         self.registered += 1
...     def setstate(self, ob):
...         ob.__setstate__({'x': 42})
```

Note: Notice that the `DM` class always sets the `x` attribute to the value `42` when activating an object.

1.3 Persistent objects without a Data Manager

Before persistent instance has been associated with a a data manager (i.e., its `_p_jar` is still `None`).

The examples below use a class, `P`, defined as:

```
>>> from persistent import Persistent
>>> from persistent.interfaces import GHOST, UPTODATE, CHANGED
>>> class P(Persistent):
...     def __init__(self):
...         self.x = 0
...     def inc(self):
...         self.x += 1
```

Instances of the derived `P` class which are not (yet) assigned to a *data manager* behave as other Python instances, except that they have some extra attributes:

```
>>> p = P()
>>> p.x
0
```

The `_p_changed` attribute is a three-state flag: it can be one of `None` (the object is not loaded), `False` (the object has not been changed since it was loaded) or `True` (the object has been changed). Until the object is assigned a *jar*, this attribute will always be `False`.

```
>>> p._p_changed
False
```

The `_p_state` attribute is an integer, representing which of the "persistent lifecycle" states the object is in. Until the object is assigned a *jar*, this attribute will always be `0` (the `UPTODATE` constant):

```
>>> p._p_state == UPTODATE
True
```

The `_p_jar` attribute is the object's *data manager*. Since it has not yet been assigned, its value is `None`:

```
>>> print(p._p_jar)
None
```

The `_p_oid` attribute is the *object id*, a unique value normally assigned by the object's *data manager*. Since the object has not yet been associated with its *jar*, its value is `None`:

```
>>> print(p._p_oid)
None
```

Without a data manager, modifying a persistent object has no effect on its `_p_state` or `_p_changed`.

```
>>> p.inc()
>>> p.inc()
>>> p.x
2
>>> p._p_changed
False
>>> p._p_state
0
```

Try all sorts of different ways to change the object's state:

```
>>> p._p_deactivate()
>>> p._p_state
0
>>> p._p_changed
False
>>> p._p_changed = True
>>> p._p_changed
False
>>> p._p_state
0
>>> del p._p_changed
>>> p._p_changed
False
>>> p._p_state
0
>>> p.x
2
```

1.4 Associating an Object with a Data Manager

Once associated with a data manager, a persistent object's behavior changes:

```
>>> p = P()
>>> dm = DM()
>>> p._p_oid = "00000012"
>>> p._p_jar = dm
>>> p._p_changed
False
>>> p._p_state
0
>>> p.__dict__
{'x': 0}
>>> dm.registered
0
```

Modifying the object marks it as changed and registers it with the data manager. Subsequent modifications don't have additional side-effects.

```
>>> p.inc()
>>> p.x
1
>>> p.__dict__
{'x': 1}
>>> p._p_changed
```

(continues on next page)

(continued from previous page)

```
True
>>> p._p_state
1
>>> dm.registered
1
>>> p.inc()
>>> p._p_changed
True
>>> p._p_state
1
>>> dm.registered
1
```

Object which register themselves with the data manager are candidates for storage to the backing store at a later point in time.

Note that mutating a non-persistent attribute of a persistent object such as a `dict` or `list` will *not* cause the containing object to be changed. Instead you can either explicitly control the state as described below, or use a *`PersistentList`* or *`PersistentMapping`*.

1.5 Explicitly controlling `_p_state`

Persistent objects expose three methods for moving an object into and out of the "ghost" state:: `persistent.Persistent._p_activate()`, `persistent.Persistent._p_deactivate()`, and `persistent.Persistent._p_invalidate()`:

```
>>> p = P()
>>> p._p_oid = '00000012'
>>> p._p_jar = DM()
```

After being assigned a jar, the object is initially in the `UPTODATE` state:

```
>>> p._p_state
0
```

From that state, `_p_deactivate` rests the object to the `GHOST` state:

```
>>> p._p_deactivate()
>>> p._p_state
-1
```

From the `GHOST` state, `_p_activate` reloads the object's data and moves it to the `UPTODATE` state:

```
>>> p._p_activate()
>>> p._p_state
0
>>> p.x
42
```

Changing the object puts it in the `CHANGED` state:

```
>>> p.inc()
>>> p.x
43
```

(continues on next page)

(continued from previous page)

```
>>> p._p_state
1
```

Attempting to deactivate in the `CHANGED` state is a no-op:

```
>>> p._p_deactivate()
>>> p.__dict__
{'x': 43}
>>> p._p_changed
True
>>> p._p_state
1
```

`_p_invalidate` forces objects into the `GHOST` state; it works even on objects in the `CHANGED` state, which is the key difference between deactivation and invalidation:

```
>>> p._p_invalidate()
>>> p.__dict__
{}
>>> p._p_state
-1
```

You can manually reset the `_p_changed` field to `False`: in this case, the object changes to the `UPTODATE` state but retains its modifications:

```
>>> p.inc()
>>> p.x
43
>>> p._p_changed = False
>>> p._p_state
0
>>> p._p_changed
False
>>> p.x
43
```

For an object in the "ghost" state, assigning `True` (or any value which is coercible to `True`) to its `_p_changed` attributes activates the object, which is exactly the same as calling `_p_activate`:

```
>>> p._p_invalidate()
>>> p._p_state
-1
>>> p._p_changed = True
>>> p._p_changed
True
>>> p._p_state
1
>>> p.x
42
```

1.6 The pickling protocol

Because persistent objects need to control how they are pickled and unpickled, the *persistent.Persistent* base class overrides the implementations of `__getstate__()` and `__setstate__()`:

```
>>> p = P()
>>> dm = DM()
>>> p._p_oid = "00000012"
>>> p._p_jar = dm
>>> p.__getstate__()
{'x': 0}
>>> p._p_state
0
```

Calling `__setstate__` always leaves the object in the uptodate state.

```
>>> p.__setstate__({'x': 5})
>>> p._p_state
0
```

A *volatile attribute* is an attribute those whose name begins with a special prefix (`_v__`). Unlike normal attributes, volatile attributes do not get stored in the object's *pickled data*.

```
>>> p._v_foo = 2
>>> p.__getstate__()
{'x': 5}
```

Assigning to volatile attributes doesn't cause the object to be marked as changed:

```
>>> p._p_state
0
```

The `_p_serial` attribute is not affected by calling setstate.

```
>>> p._p_serial = b"00000012"
>>> p.__setstate__(p.__getstate__())
>>> p._p_serial
b'00000012'
```

1.7 Estimated Object Size

We can store a size estimation in `_p_estimated_size`. Its default is 0. The size estimation can be used by a cache associated with the data manager to help in the implementation of its replacement strategy or its size bounds.

```
>>> p._p_estimated_size
0
>>> p._p_estimated_size = 1000
>>> p._p_estimated_size
1024
```

Huh? Why is the estimated size coming out different than what we put in? The reason is that the size isn't stored exactly. For backward compatibility reasons, the size needs to fit in 24 bits, so, internally, it is adjusted somewhat.

Of course, the estimated size must not be negative.

```
>>> p._p_estimated_size = -1
Traceback (most recent call last):
...
ValueError: _p_estimated_size must not be negative
```

1.8 Overriding the attribute protocol

Subclasses which override the attribute-management methods provided by *persistent.Persistent*, but must obey some constraints:

__getattribute__() When overriding __getattribute__, the derived class implementation **must** first call persistent.IPersistent._p_getattr(), passing the name being accessed. This method ensures that the object is activated, if needed, and handles the "special" attributes which do not require activation (e.g., _p_oid, __class__, __dict__, etc.) If _p_getattr returns True, the derived class implementation **must** delegate to the base class implementation for the attribute.

__setattr__() When overriding __setattr__, the derived class implementation **must** first call persistent.IPersistent._p_setattr(), passing the name being accessed and the value. This method ensures that the object is activated, if needed, and handles the "special" attributes which do not require activation (_p_*). If _p_setattr returns True, the derived implementation must assume that the attribute value has been set by the base class.

__delattr__() When overriding __delattr__, the derived class implementation **must** first call persistent.IPersistent._p_delattr(), passing the name being accessed. This method ensures that the object is activated, if needed, and handles the "special" attributes which do not require activation (_p_*). If _p_delattr returns True, the derived implementation must assume that the attribute has been deleted base class.

__getattr__() For the __getattr__ method, the behavior is like that for regular Python classes and for earlier versions of ZODB 3.

1.9 Implementing _p_repr

Subclasses can implement _p_repr to provide a custom representation. If this method raises an exception, the default representation will be used. The benefit of implementing _p_repr instead of overriding __repr__ is that it provides safer handling for objects that can't be activated because their persistent data is missing or their jar is closed.

```
>>> class P(Persistent):
...     def _p_repr(self):
...         return "Custom repr"

>>> p = P()
>>> print(repr(p))
Custom repr
```

CHAPTER 2

`persistent` API documentation

2.1 `persistent.interfaces`

Persistence Interfaces

interface `persistent.interfaces.`**IPersistent**

Python persistent interface

A persistent object can be in one of several states:

- Unsaved

  The object has been created but not saved in a data manager.

  In this state, the _p_changed attribute is non-None and false and the _p_jar attribute is None.

- Saved

  The object has been saved and has not been changed since it was saved.

  In this state, the _p_changed attribute is non-None and false and the _p_jar attribute is set to a data manager.

- Sticky

  This state is identical to the saved state except that the object cannot transition to the ghost state. This is a special state used by C methods of persistent objects to make sure that state is not unloaded in the middle of computation.

  In this state, the _p_changed attribute is non-None and false and the _p_jar attribute is set to a data manager.

  There is no Python API for detecting whether an object is in the sticky state.

- Changed

  The object has been changed.

  In this state, the _p_changed attribute is true and the _p_jar attribute is set to a data manager.

- Ghost

the object is in memory but its state has not been loaded from the database (or its state has been unloaded). In this state, the object doesn't contain any application data.

In this state, the _p_changed attribute is None, and the _p_jar attribute is set to the data manager from which the object was obtained.

In all the above, _p_oid (the persistent object id) is set when _p_jar first gets set.

The following state transitions are possible:

- Unsaved -> Saved

  This transition occurs when an object is saved in the database. This usually happens when an unsaved object is added to (e.g. as an attribute or item of) a saved (or changed) object and the transaction is committed.

- Saved -> Changed Sticky -> Changed Ghost -> Changed

  This transition occurs when someone sets an attribute or sets _p_changed to a true value on a saved, sticky or ghost object. When the transition occurs, the persistent object is required to call the register() method on its data manager, passing itself as the only argument.

  Prior to ZODB 3.6, setting _p_changed to a true value on a ghost object was ignored (the object remained a ghost, and getting its _p_changed attribute continued to return None).

- Saved -> Sticky

  This transition occurs when C code marks the object as sticky to prevent its deactivation.

- Saved -> Ghost

  This transition occurs when a saved object is deactivated or invalidated. See discussion below.

- Sticky -> Saved

  This transition occurs when C code unmarks the object as sticky to allow its deactivation.

- Changed -> Saved

  This transition occurs when a transaction is committed. After saving the state of a changed object during transaction commit, the data manager sets the object's _p_changed to a non-None false value.

- Changed -> Ghost

  This transition occurs when a transaction is aborted. All changed objects are invalidated by the data manager by an abort.

- Ghost -> Saved

  This transition occurs when an attribute or operation of a ghost is accessed and the object's state is loaded from the database.

Note that there is a separate C API that is not included here. The C API requires a specific data layout and defines the sticky state.

About Invalidation, Deactivation and the Sticky & Ghost States

The sticky state is intended to be a short-lived state, to prevent an object's state from being discarded while we're in C routines. It is an error to invalidate an object in the sticky state.

Deactivation is a request that an object discard its state (become a ghost). Deactivation is an optimization, and a request to deactivate may be ignored. There are two equivalent ways to request deactivation:

- call _p_deactivate()
- set _p_changed to None

There are two ways to invalidate an object: call the _p_invalidate() method (preferred) or delete its _p_changed attribute. This cannot be ignored, and is used when semantics require invalidation. Normally, an invalidated object transitions to the ghost state. However, some objects cannot be ghosts. When these objects are invalidated, they immediately reload their state from their data manager, and are then in the saved state.

reprs

By default, persistent objects include the reprs of their _p_oid and _p_jar, if any, in their repr. If a subclass implements the optional method `_p_repr`, it will be called and its results returned instead of the default repr; if this method raises an exception, that exception will be caught and its repr included in the default repr.

_p_jar
The data manager for the object.

The data manager should implement IPersistentDataManager (note that this constraint is not enforced).

If there is no data manager, then this is None.

Once assigned to a data manager, an object cannot be re-assigned to another.

_p_oid
The object id.

It is up to the data manager to assign this.

The special value None is reserved to indicate that an object id has not been assigned. Non-None object ids must be non-empty strings. The 8-byte string consisting of 8 NUL bytes (‘’) is reserved to identify the database root object.

Once assigned an OID, an object cannot be re-assigned another.

_p_changed
The persistent state of the object.

This is one of:

None – The object is a ghost.

false but not None – The object is saved (or has never been saved).

true – The object has been modified since it was last saved.

The object state may be changed by assigning or deleting this attribute; however, assigning None is ignored if the object is not in the saved state, and may be ignored even if the object is in the saved state.

At and after ZODB 3.6, setting _p_changed to a true value for a ghost object activates the object; prior to 3.6, setting _p_changed to a true value on a ghost object was ignored.

Note that an object can transition to the changed state only if it has a data manager. When such a state change occurs, the ‘register’ method of the data manager must be called, passing the persistent object.

Deleting this attribute forces invalidation independent of existing state, although it is an error if the sticky state is current.

_p_serial
The object serial number.

This member is used by the data manager to distiguish distinct revisions of a given persistent object.

This is an 8-byte string (not Unicode).

_p_mtime
The object’s modification time (read-only).

This is a float, representing seconds since the epoch (as returned by time.time).

_p_state
The object's persistence state token.

Must be one of GHOST, UPTODATE, CHANGED, or STICKY.

_p_estimated_size
An estimate of the object's size in bytes.

May be set by the data manager.

__getattribute__(*name*)
Handle activating ghosts before returning an attribute value.

"Special" attributes and '_p_*' attributes don't require activation.

__setattr__(*name*, *value*)
Handle activating ghosts before setting an attribute value.

"Special" attributes and '_p_*' attributes don't require activation.

__delattr__(*name*)
Handle activating ghosts before deleting an attribute value.

"Special" attributes and '_p_*' attributes don't require activation.

__getstate__()
Get the object data.

The state should not include persistent attributes ("_p_name"). The result must be picklable.

__setstate__(*state*)
Set the object data.

__reduce__()
Reduce an object to contituent parts for serialization.

_p_activate()
Activate the object.

Change the object to the saved state if it is a ghost.

_p_deactivate()
Deactivate the object.

Possibly change an object in the saved state to the ghost state. It may not be possible to make some persistent objects ghosts, and, for optimization reasons, the implementation may choose to keep an object in the saved state.

_p_invalidate()
Invalidate the object.

Invalidate the object. This causes any data to be thrown away, even if the object is in the changed state. The object is moved to the ghost state; further accesses will cause object data to be reloaded.

_p_getattr(*name*)
Test whether the base class must handle the name

The method unghostifies the object, if necessary. The method records the object access, if necessary.

This method should be called by subclass __getattribute__ implementations before doing anything else. If the method returns True, then __getattribute__ implementations must delegate to the base class, Persistent.

_p_setattr(*name*, *value*)
Save persistent meta data

This method should be called by subclass __setattr__ implementations before doing anything else. If it returns true, then the attribute was handled by the base class.

The method unghostifies the object, if necessary. The method records the object access, if necessary.

_p_delattr(*name*)
Delete persistent meta data

This method should be called by subclass __delattr__ implementations before doing anything else. If it returns true, then the attribute was handled by the base class.

The method unghostifies the object, if necessary. The method records the object access, if necessary.

interface persistent.interfaces.**IPersistentDataManager**
Provide services for managing persistent state.

This interface is used by a persistent object to interact with its data manager in the context of a transaction.

_cache
The pickle cache associated with this connection.

setstate(*object*)
Load the state for the given object.

The object should be in the ghost state. The object's state will be set and the object will end up in the saved state.

The object must provide the IPersistent interface.

oldstate(*obj*, *tid*)
Return copy of 'obj' that was written by transaction 'tid'.

The returned object does not have the typical metadata (_p_jar, _p_oid, _p_serial) set. I'm not sure how references to other peristent objects are handled.

Parameters obj: a persistent object from this Connection. tid: id of a transaction that wrote an earlier revision.

Raises KeyError if tid does not exist or if tid deleted a revision of obj.

register(*object*)
Register an IPersistent with the current transaction.

This method must be called when the object transitions to the changed state.

A subclass could override this method to customize the default policy of one transaction manager for each thread.

interface persistent.interfaces.**IPickleCache**
API of the cache for a ZODB connection.

__getitem__(*oid*)
-> the persistent object for OID.

o Raise KeyError if not found.

__setitem__(*oid*, *value*)
Save the persistent object under OID.

o 'oid' must be a string, else raise ValueError.

o Raise KeyError on duplicate

__delitem__(*oid*)
Remove the persistent object for OID.

o ‘oid’ must be a string, else raise ValueError.

o Raise KeyError if not found.

get(*oid*, *default=None*)
-> the persistent object for OID.

o Return ‘default’ if not found.

__len__()
-> the number of OIDs in the cache.

items()
-> a sequence of tuples (oid, value) for cached objects.

o Only includes items in ‘data’ (no p-classes).

ringlen()
-> the number of persistent objects in the ring.

o Only includes items in the ring (no ghosts or p-classes).

lru_items()
-> a sequence of tuples (oid, value) for cached objects.

o Tuples will be in LRU order.

o Only includes items in the ring (no ghosts or p-classes).

klass_items()
-> a sequence of tuples (oid, value) for cached p-classes.

o Only includes persistent classes.

incrgc()
Perform an incremental garbage collection sweep.

o Reduce number of non-ghosts to ‘cache_size’, if possible.

o Ghostify in LRU order.

o Skip dirty or sticky objects.

o Quit once we get down to ‘cache_size’.

full_sweep()
Perform a full garbage collection sweep.

o Reduce number of non-ghosts to 0, if possible.

o Ghostify all non-sticky / non-changed objecs.

minimize()
Alias for ‘full_sweep’.

o XXX?

new_ghost(*oid*, *obj*)
Add the given (ghost) object to the cache.

Also, set its _p_jar and _p_oid, and ensure it is in the GHOST state.

If the object doesn’t define ‘_p_oid’ / ‘_p_jar’, raise.

If the object’s ‘_p_oid’ is not None, raise.

If the object’s ‘_p_jar’ is not None, raise.

If 'oid' is already in the cache, raise.

invalidate(*to_invalidate*)
Invalidate the indicated objects.

o If 'to_invalidate' is a string, treat it as an OID.

o Otherwise, iterate over it as a sequence of OIDs.

o Any OID corresponding to a p-class will cause the corresponding p-class to be removed from the cache.

o For all other OIDs, ghostify the corrsponding object and remove it from the ring.

debug_info()
Return debugging data about objects in the cache.

o Return a sequence of tuples, (oid, refcount, typename, state).

update_object_size_estimation(*oid*, *new_size*)
Update the cache's size estimation for 'oid', if known to the cache.

cache_size
Target size of the cache

cache_drain_resistance
Factor for draining cache below target size

cache_non_ghost_count
Number of non-ghosts in the cache (XXX how is it different from ringlen?

cache_data
Property: copy of our 'data' dict

cache_klass_count
Property: len of 'persistent_classes'

2.2 Implementations

This package provides one implementation of *IPersistent* that should be extended.

class persistent.**Persistent**
Bases: object

2.3 Persistent Collections

The persistent package provides two simple collections that are persistent and keep track of when they are mutated in place.

class persistent.mapping.**PersistentMapping**(***kwargs*)
Bases: UserDict.IterableUserDict, *persistent.Persistent*

A persistent wrapper for mapping objects.

This class allows wrapping of mapping objects so that object changes are registered. As a side effect, mapping objects may be subclassed.

A subclass of PersistentMapping or any code that adds new attributes should not create an attribute named _container. This is reserved for backwards compatibility reasons.

clear()
Remove all data from this dictionary.

Changed in version 4.5.2: If there was nothing to remove, this object is no longer marked as modified.

popitem()
Remove an item.

Changed in version 4.5.2: No longer marks this object as modified if it was empty and an exception raised.

update([*E*], ***F*) → None.
Changed in version 4.5.2: Now accepts arbitrary keyword arguments. In the special case of a keyword argument named `b` that is a dictionary, the behaviour will change.

class persistent.list.**PersistentList**(*initlist=None*)
Bases: UserList.UserList, *persistent.Persistent*

A persistent wrapper for list objects.

Mutating instances of this class will cause them to be marked as changed and automatically persisted.

Changed in version 4.5.2: Using the *clear* method, or deleting a slice (e.g., `del inst[:]` or `del inst[x:x]`) now only results in marking the instance as changed if it actually removed items.

Changed in version 4.5.2: The *copy* method is available on Python 2.

append(*item*)
S.append(object) – append object to the end of the sequence

clear()
Remove all items from the list.

Changed in version 4.5.2: Now marks the list as changed, and is available on both Python 2 and Python 3.

extend(*other*)
S.extend(iterable) – extend sequence by appending elements from the iterable

insert(*i*, *item*)
S.insert(index, object) – insert object before index

pop([*index*]) → item – remove and return item at index (default last).
Raise IndexError if list is empty or index is out of range.

remove(*item*)
S.remove(value) – remove first occurrence of value. Raise ValueError if the value is not present.

reverse()
S.reverse() – reverse *IN PLACE*

2.4 Customizing Attribute Access

2.4.1 Hooking `__getattr__()`

The __getattr__ method works pretty much the same for persistent classes as it does for other classes. No special handling is needed. If an object is a ghost, then it will be activated before __getattr__ is called.

In this example, our objects returns a tuple with the attribute name, converted to upper case and the value of _p_changed, for any attribute that isn't handled by the default machinery.

```
>>> from persistent.tests.attrhooks import OverridesGetattr
>>> o = OverridesGetattr()
>>> o._p_changed
False
>>> o._p_oid
>>> o._p_jar
>>> o.spam
('SPAM', False)
>>> o.spam = 1
>>> o.spam
1
```

We'll save the object, so it can be deactivated:

```
>>> from persistent.tests.attrhooks import _resettingJar
>>> jar = _resettingJar()
>>> jar.add(o)
>>> o._p_deactivate()
>>> o._p_changed
```

And now, if we ask for an attribute it doesn't have,

```
>>> o.eggs
('EGGS', False)
```

And we see that the object was activated before calling the `__getattr__()` method.

2.4.2 Hooking All Access

In this example, we'll provide an example that shows how to override the `__getattribute__()`, `__setattr__()`, and `__delattr__()` methods. We'll create a class that stores it's attributes in a secret dictionary within the instance dictionary.

The class will have the policy that variables with names starting with `tmp_` will be volatile.

Our sample class takes initial values as keyword arguments to the constructor:

```
>>> from persistent.tests.attrhooks import VeryPrivate
>>> o = VeryPrivate(x=1)
```

Hooking `__getattribute__`()

The `__getattribute__()` method is called for all attribute accesses. It overrides the attribute access support inherited from Persistent.

```
>>> o._p_changed
False
>>> o._p_oid
>>> o._p_jar
>>> o.x
1
>>> o.y
Traceback (most recent call last):
...
AttributeError: y
```

Next, we'll save the object in a database so that we can deactivate it:

```
>>> from persistent.tests.attrhooks import _rememberingJar
>>> jar = _rememberingJar()
>>> jar.add(o)
>>> o._p_deactivate()
>>> o._p_changed
```

And we'll get some data:

```
>>> o.x
1
```

which activates the object:

```
>>> o._p_changed
False
```

It works for missing attributes too:

```
>>> o._p_deactivate()
>>> o._p_changed

>>> o.y
Traceback (most recent call last):
...
AttributeError: y

>>> o._p_changed
False
```

Hooking `__setattr__()`

The `__setattr__()` method is called for all attribute assignments. It overrides the attribute assignment support inherited from Persistent.

Implementors of `__setattr__()` methods:

1. Must call Persistent._p_setattr first to allow it to handle some attributes and to make sure that the object is activated if necessary, and
2. Must set _p_changed to mark objects as changed.

```
>>> o = VeryPrivate()
>>> o._p_changed
False
>>> o._p_oid
>>> o._p_jar
>>> o.x
Traceback (most recent call last):
...
AttributeError: x

>>> o.x = 1
>>> o.x
1
```

Because the implementation doesn't store attributes directly in the instance dictionary, we don't have a key for the attribute:

```
>>> 'x' in o.__dict__
False
```

Next, we'll give the object a "remembering" jar so we can deactivate it:

```
>>> jar = _rememberingJar()
>>> jar.add(o)
>>> o._p_deactivate()
>>> o._p_changed
```

We'll modify an attribute

```
>>> o.y = 2
>>> o.y
2
```

which reactivates it, and marks it as modified, because our implementation marked it as modified:

```
>>> o._p_changed
True
```

Now, if fake a commit:

```
>>> jar.fake_commit()
>>> o._p_changed
False
```

And deactivate the object:

```
>>> o._p_deactivate()
>>> o._p_changed
```

and then set a variable with a name starting with `tmp_`, The object will be activated, but not marked as modified, because our `__setattr__()` implementation doesn't mark the object as changed if the name starts with `tmp_`:

```
>>> o.tmp_foo = 3
>>> o._p_changed
False
>>> o.tmp_foo
3
```

Hooking `__delattr__`()

The __delattr__ method is called for all attribute deletions. It overrides the attribute deletion support inherited from Persistent.

Implementors of `__delattr__()` methods:

1. Must call Persistent._p_delattr first to allow it to handle some attributes and to make sure that the object is activated if necessary, and
2. Must set _p_changed to mark objects as changed.

```
>>> o = VeryPrivate(x=1, y=2, tmp_z=3)
>>> o._p_changed
False
>>> o._p_oid
>>> o._p_jar
>>> o.x
1
>>> del o.x
>>> o.x
Traceback (most recent call last):
...
AttributeError: x
```

Next, we'll save the object in a jar so that we can deactivate it:

```
>>> jar = _rememberingJar()
>>> jar.add(o)
>>> o._p_deactivate()
>>> o._p_changed
```

If we delete an attribute:

```
>>> del o.y
```

The object is activated. It is also marked as changed because our implementation marked it as changed.

```
>>> o._p_changed
True
>>> o.y
Traceback (most recent call last):
...
AttributeError: y

>>> o.tmp_z
3
```

Now, if fake a commit:

```
>>> jar.fake_commit()
>>> o._p_changed
False
```

And deactivate the object:

```
>>> o._p_deactivate()
>>> o._p_changed
```

and then delete a variable with a name starting with `tmp_`, The object will be activated, but not marked as modified, because our `__delattr__()` implementation doesn't mark the object as changed if the name starts with `tmp_`:

```
>>> del o.tmp_z
>>> o._p_changed
False
>>> o.tmp_z
Traceback (most recent call last):
...
AttributeError: tmp_z
```

If we attempt to delete _p_oid, we find that we can't, and the object is also not activated or changed:

```
>>> del o._p_oid
Traceback (most recent call last):
...
ValueError: can't delete _p_oid of cached object
>>> o._p_changed
False
```

We are allowed to delete _p_changed, which sets it to None:

```
>>> del o._p_changed
>>> o._p_changed is None
True
```

2.5 Pickling Persistent Objects

Persistent objects are designed to make the standard Python pickling machinery happy:

```
>>> import pickle
>>> from persistent.tests.cucumbers import Simple
>>> from persistent.tests.cucumbers import print_dict

>>> x = Simple('x', aaa=1, bbb='foo')

>>> print_dict(x.__getstate__())
{'__name__': 'x', 'aaa': 1, 'bbb': 'foo'}

>>> f, (c,), state = x.__reduce__()
>>> f.__name__
'__newobj__'
>>> f.__module__.replace('_', '') # Normalize Python2/3
'copyreg'
>>> c.__name__
'Simple'

>>> print_dict(state)
{'__name__': 'x', 'aaa': 1, 'bbb': 'foo'}

>>> import pickle
>>> pickle.loads(pickle.dumps(x)) == x
True
>>> pickle.loads(pickle.dumps(x, 0)) == x
True
>>> pickle.loads(pickle.dumps(x, 1)) == x
True

>>> pickle.loads(pickle.dumps(x, 2)) == x
True

>>> x.__setstate__({'z': 1})
>>> x.__dict__
{'z': 1}
```

This support even works well for derived classes which customize pickling by overriding __getnewargs__(), __getstate__() and __setstate__().

```
>>> from persistent.tests.cucumbers import Custom

>>> x = Custom('x', 'y')
>>> x.__getnewargs__()
('x', 'y')
>>> x.a = 99

>>> (f, (c, ax, ay), a) = x.__reduce__()
>>> f.__name__
'__newobj__'
>>> f.__module__.replace('_', '') # Normalize Python2/3
'copyreg'
>>> c.__name__
'Custom'
>>> ax, ay, a
('x', 'y', 99)

>>> pickle.loads(pickle.dumps(x)) == x
True
>>> pickle.loads(pickle.dumps(x, 0)) == x
True
>>> pickle.loads(pickle.dumps(x, 1)) == x
True
>>> pickle.loads(pickle.dumps(x, 2)) == x
True
```

The support works for derived classes which define `__slots__`. It ignores any slots which map onto the "persistent" namespace (prefixed with `_p_`) or the "volatile" namespace (prefixed with `_v_`):

```
>>> from persistent.tests.cucumbers import SubSlotted
>>> x = SubSlotted('x', 'y', 'z')
```

Note that we haven't yet assigned a value to the `s4` attribute:

```
>>> d, s = x.__getstate__()
>>> d
>>> print_dict(s)
{'s1': 'x', 's2': 'y', 's3': 'z'}

>>> import pickle
>>> pickle.loads(pickle.dumps(x)) == x
True
>>> pickle.loads(pickle.dumps(x, 0)) == x
True
>>> pickle.loads(pickle.dumps(x, 1)) == x
True
>>> pickle.loads(pickle.dumps(x, 2)) == x
True
```

After assigning it:

```
>>> x.s4 = 'spam'

>>> d, s = x.__getstate__()
>>> d
>>> print_dict(s)
{'s1': 'x', 's2': 'y', 's3': 'z', 's4': 'spam'}
```

(continues on next page)

(continued from previous page)

```
>>> pickle.loads(pickle.dumps(x)) == x
True
>>> pickle.loads(pickle.dumps(x, 0)) == x
True
>>> pickle.loads(pickle.dumps(x, 1)) == x
True
>>> pickle.loads(pickle.dumps(x, 2)) == x
True
```

persistent.Persistent supports derived classes which have base classes defining `__slots`, but which do not define attr:*__slots__* themselves:

```
>>> from persistent.tests.cucumbers import SubSubSlotted
>>> x = SubSubSlotted('x', 'y', 'z')

>>> d, s = x.__getstate__()
>>> print_dict(d)
{}
>>> print_dict(s)
{'s1': 'x', 's2': 'y', 's3': 'z'}

>>> import pickle
>>> pickle.loads(pickle.dumps(x)) == x
True
>>> pickle.loads(pickle.dumps(x, 0)) == x
True
>>> pickle.loads(pickle.dumps(x, 1)) == x
True
>>> pickle.loads(pickle.dumps(x, 2)) == x
True

>>> x.s4 = 'spam'
>>> x.foo = 'bar'
>>> x.baz = 'bam'

>>> d, s = x.__getstate__()
>>> print_dict(d)
{'baz': 'bam', 'foo': 'bar'}
>>> print_dict(s)
{'s1': 'x', 's2': 'y', 's3': 'z', 's4': 'spam'}

>>> pickle.loads(pickle.dumps(x)) == x
True
>>> pickle.loads(pickle.dumps(x, 0)) == x
True
>>> pickle.loads(pickle.dumps(x, 1)) == x
True
>>> pickle.loads(pickle.dumps(x, 2)) == x
True
```

2.6 Caching Persistent Objects

2.6.1 Creating Objects de novo

Creating ghosts from scratch, as opposed to ghostifying a non-ghost is rather tricky. `IPeristent` doesn't really provide the right interface given that:

- `_p_deactivate()` and `_p_invalidate()` are overridable, and could assume that the object's state is properly initialized.
- Assigning `_p_changed` to None just calls `_p_deactivate()`.
- Deleting `_p_changed` just calls `_p_invalidate()`.

Note: The current cache implementation is intimately tied up with the persistence implementation and has internal access to the persistence state. The cache implementation can update the persistence state for newly created and uninitialized objects directly.

The future persistence and cache implementations will be far more decoupled. The persistence implementation will only manage object state and generate object-usage events. The cache implementation(s) will be responsible for managing persistence-related (meta-)state, such as _p_state, _p_changed, _p_oid, etc. So in that future implementation, the cache will be more central to managing object persistence information.

Caches have a `new_ghost()` method that:

- adds an object to the cache, and
- initializes its persistence data.

```
>>> import persistent
>>> from persistent.tests.utils import ResettingJar

>>> class C(persistent.Persistent):
...     pass

>>> jar = ResettingJar()
>>> cache = persistent.PickleCache(jar, 10, 100)
>>> ob = C.__new__(C)
>>> cache.new_ghost(b'1', ob)

>>> ob._p_changed
>>> ob._p_jar is jar
True
>>> ob._p_oid == b'1'
True

>>> cache.cache_non_ghost_count
0
```

CHAPTER 3

`persistent` Changelog

3.1 4.9.1 (2022-09-16)

- Update Python 3.11 support to 3.11.0-rc1.
- Disable unsafe math optimizations in C code. See pull request 176.

3.2 4.9.0 (2022-03-10)

- Add support for Python 3.11 (as of 3.11a5).

3.3 4.8.0 (2022-03-07)

- Switch package to src-layout, this is a packaging only change. (#168)
- Add support for Python 3.10.

3.4 4.7.0 (2021-04-13)

- Add support for Python 3.9.
- Move from Travis CI to Github Actions.
- Supply manylinux wheels for aarch64 (ARM).
- Fix the pure-Python implementation to activate a ghost object when setting its `__class__` and `__dict__`. This matches the behaviour of the C implementation. See issue 155.
- Fix the CFFI cache implementation (used on CPython when `PURE_PYTHON=1`) to not print unraisable `AttributeErrors` from `_WeakValueDictionary` during garbage collection. See issue 150.

- Make the pure-Python implementation of the cache run a garbage collection (`gc.collect()`) on `full_sweep`, `incrgc` and `minimize` *if* it detects that an object that was weakly referenced has been ejected. This solves issues on PyPy with ZODB raising `ConnectionStateError` when there are persistent `zope.interface` utilities/adapters registered. This partly reverts a change from release 4.2.3.

3.5 4.6.4 (2020-03-26)

- Fix an overly specific test failure using zope.interface 5. See issue 144.
- Fix two reference leaks that could theoretically occur as the result of obscure errors. See issue 143.

3.6 4.6.3 (2020-03-18)

- Fix a crash in the test suite under a 32-bit CPython on certain 32-bit platforms. See issue 137. Fix by Jerry James.

3.7 4.6.2 (2020-03-12)

- Fix an `AssertionError` clearing a non-empty `PersistentMapping` that has no connection. See issue 139.

3.8 4.6.1 (2020-03-06)

- Stop installing C header files on PyPy (which is what persistent before 4.6.0 used to do), fixes issue 135.

3.9 4.6.0 (2020-03-05)

- Fix slicing of `PersistentList` to always return instances of the same class. It was broken on Python 3 prior to 3.7.4.
- Fix copying of `PersistentList` and `PersistentMapping` using `copy.copy` to also copy the underlying data object. This was broken prior to Python 3.7.4.
- Update the handling of the `PURE_PYTHON` environment variable. Now, a value of "0" requires that the C extensions be used; any other non-empty value prevents the extensions from being used. Also, all C extensions are required together or none of them will be used. This prevents strange errors that arise from a mismatch of Python and C implementations. See issue 131.

  Note that some private implementation details such as the names of the pure-Python implementations have changed.
- Fix `PersistentList` to mark itself as changed after calling `clear` (if needed). See PR 115.
- Fix `PersistentMapping.update` to accept keyword arguments like the native `UserDict`. Previously, most uses of keyword arguments resulted in `TypeError`; in the undocumented and extremely unlikely event of a single keyword argument called `b` that happens to be a dictionary, the behaviour will change. Also adjust the signatures of `setdefault` and `pop` to match the native version.

- Fix `PersistentList.clear`, `PersistentMapping.clear` and `PersistentMapping.popitem` to no longer mark the object as changed if it was empty.
- Add preliminary support for Python 3.9a3+. See issue 124.
- Fix the Python implementation of the PickleCache to be able to store objects that cannot be weakly referenced. See issue 133.

  Note that `ctypes` is required to use the Python implementation (except on PyPy).

3.10 4.5.1 (2019-11-06)

- Add support for Python 3.8.
- Update documentation to Python 3.

3.11 4.5.0 (2019-05-09)

- Fully test the C implementation of the PickleCache, and fix discrepancies between it and the Python implementation:
  - The C implementation now raises `ValueError` instead of `AssertionError` for certain types of bad inputs.
  - The Python implementation uses the C wording for error messages.
  - The C implementation properly implements `IPickleCache`; methods unique to the Python implementation were moved to `IExtendedPickleCache`.
  - The Python implementation raises `AttributeError` if a persistent class doesn't have a `p_jar` attribute.

  See issue 102.
- Allow sweeping cache without `cache_size`. `cache_size_bytes` works with `cache_size=0`, no need to set `cache_size` to a large value.
- Require `CFFI` on CPython for pure-Python operation. This drops support for Jython (which was untested). See issue 77.
- Fix DeprecationWarning about `PY_SSIZE_T_CLEAN`. See issue 108.
- Drop support for Python 3.4.

3.12 4.4.3 (2018-10-22)

- Fix the repr of the persistent objects to include the module name when using the C extension. This matches the pure-Python behaviour and the behaviour prior to 4.4.0. See issue 92.
- Change the repr of persistent objects to format the OID as in integer in hexadecimal notation if it is an 8-byte byte string, as ZODB does. This eliminates some issues in doctests. See issue 95.

3.13 4.4.2 (2018-08-28)

- Explicitly use unsigned constants for packing and unpacking C timestamps, fixing an arithmetic issue for GCC when optimizations are enabled and `-fwrapv` is *not* enabled. See issue 86.

3.14 4.4.1 (2018-08-23)

- Fix installation of source packages on PyPy. See issue 88.

3.15 4.4.0 (2018-08-22)

- Use unsigned constants when doing arithmetic on C timestamps, possibly avoiding some overflow issues with some compilers or compiler settings. See issue 86.
- Change the default representation of `Persistent` objects to include the representation of their OID and jar, if set. Also add the ability for subclasses to implement `_p_repr()` instead of overriding `__repr__` for better exception handling. See issue 11.
- Reach and maintain 100% test coverage.
- Simplify `__init__.py`, including removal of an attempted legacy import of `persistent.TimeStamp`. See PR 80.
- Add support for Python 3.7 and drop support for Python 3.3.
- Build the CFFI modules (used on PyPy or when PURE_PYTHON is set) at installation or wheel building time when CFFI is available. This replaces the deprecated way of building them at import time. If binary wheels are distributed, it eliminates the need to have a functioning C compiler to use PyPy. See issue 75.
- Fix deleting the `_p_oid` of a pure-Python persistent object when it is in a cache.
- Fix deleting special (`_p`) attributes of a pure-Python persistent object that overrides `__delattr__` and correctly calls `_p_delattr`.
- Remove some internal compatibility shims that are no longer necessary. See PR 82.
- Make the return value of `TimeStamp.second()` consistent across C and Python implementations when the `TimeStamp` was created from 6 arguments with floating point seconds. Also make it match across trips through `TimeStamp.raw()`. Previously, the C version could initially have erroneous rounding and too much false precision, while the Python version could have too much precision. The raw/repr values have not changed. See issue 41.

3.16 4.3.0 (2018-07-30)

- Fix the possibility of a rare crash in the C extension when deallocating items. See https://github.com/zopefoundation/persistent/issues/66
- Change cPickleCache's comparison of object sizes to determine whether an object can go in the cache to use `PyObject_TypeCheck()`. This matches what the pure Python implementation does and is a stronger test that the object really is compatible with the cache. Previously, an object could potentially include `cPersistent_HEAD` and *not* set `tp_base` to `cPersistenceCAPI->pertype` and still be eligible for the pickle cache; that is no longer the case. See issue 69.

3.17 4.2.4.2 (2017-04-23)

- Packaging-only release: fix Python 2.7 `manylinux` wheels.

3.18 4.2.4.1 (2017-04-21)

- Packaging-only release: get `manylinux` wheel built automatically.

3.19 4.2.4 (2017-03-20)

- Avoid raising a `SystemError: error return without exception set` when loading an object with slots whose jar generates an exception (such as a ZODB `POSKeyError`) in `setstate`.

3.20 4.2.3 (2017-03-08)

- Fix the hashcode of Python `TimeStamp` objects on 64-bit Python on Windows. See https://github.com/zopefoundation/persistent/pull/55
- Stop calling `gc.collect` every time `PickleCache.incrgc` is called (every transaction boundary) in pure-Python mode (PyPy). This means that the reported size of the cache may be wrong (until the next GC), but it is much faster. This should not have any observable effects for user code.
- Stop clearing the dict and slots of objects added to `PickleCache.new_ghost` (typically these values are passed to `__new__` from the pickle data) in pure-Python mode (PyPy). This matches the behaviour of the C code.
- Add support for Python 3.6.
- Fix `__setstate__` interning when `state` parameter is not a built-in dict

3.21 4.2.2 (2016-11-29)

- Drop use of `ctypes` for determining maximum integer size, to increase pure-Python compatibility. See https://github.com/zopefoundation/persistent/pull/31
- Ensure that `__slots__` attributes are cleared when a persistent object is ghostified. (This excluses classes that override `__new__`. See https://github.com/zopefoundation/persistent/wiki/Notes_on_state_new_and_slots if you're curious.)

3.22 4.2.1 (2016-05-26)

- Fix the hashcode of C `TimeStamp` objects on 64-bit Python 3 on Windows.

3.23 4.2.0 (2016-05-05)

- Fixed the Python(/PYPY) implementation `TimeStamp.timeTime` method to have subsecond precision.
- When testing `PURE_PYTHON` environments under `tox`, avoid poisoning the user's global wheel cache.
- Add support for Python 3.5.
- Drop support for Python 2.6 and 3.2.

3.24 4.1.1 (2015-06-02)

- Fix manifest and re-upload to fix stray files included in 4.1.0.

3.25 4.1.0 (2015-05-19)

- Make the Python implementation of `Persistent` and `PickleCache` behave more similarly to the C implementation. In particular, the Python version can now run the complete ZODB and ZEO test suites.
- Fix the hashcode of the Python `TimeStamp` on 32-bit platforms.

3.26 4.0.9 (2015-04-08)

- Make the C and Python `TimeStamp` objects behave more alike. The Python version now produces the same `repr` and `.raw()` output as the C version, and has the same hashcode. In addition, the Python version is now supports ordering and equality like the C version.
- Intern keys of object state in `__setstate__` to reduce memory usage when unpickling multiple objects with the same attributes.
- Add support for PyPy3.
- 100% branch coverage.

3.27 4.0.8 (2014-03-20)

- Add support for Python 3.4.
- In pure-Python `Persistent`, avoid loading state in `_p_activate` for non-ghost objects (which could corrupt their state). (PR #9)
- In pure-Python, and don't throw `POSKeyError` if `_p_activate` is called on an object that has never been committed. (PR #9)
- In pure-Python `Persistent`, avoid calling a subclass's `__setattr__` at instance creation time. (PR #8)
- Make it possible to delete `_p_jar` / `_p_oid` of a pure-Python `Persistent` object which has been removed from the jar's cache (fixes aborting a ZODB Connection that has added objects). (PR #7)

3.28 4.0.7 (2014-02-20)

- Avoid a KeyError from `_p_accessed()` on newly-created objects under pure-Python: these objects may be assigned to a jar, but not yet added to its cache. (PR #6)
- Avoid a failure in `Persistent.__setstate__` when the state dict contains exactly two keys. (PR #5)
- Fix a hang in `picklecache` invalidation if OIDs are manually passed out-of-order. (PR #4)
- Add `PURE_PYTHON` environment variable support: if set, the C extensions will not be built, imported, or tested.

3.29 4.0.6 (2013-01-03)

- Updated Trove classifiers.

3.30 4.0.5 (2012-12-14)

- Fixed the C-extensions under Py3k (previously they compiled but were not importable).

3.31 4.0.4 (2012-12-11)

- Added support for Python 3.3.
- C extenstions now build under Python 3.2, passing the same tests as the pure-Python reference implementation.

3.32 4.0.3 (2012-11-19)

- Fixed: In the C implimentation, an integer was compared with a pointer, with undefined results and a compiler warning.
- Fixed: the Python implementation of the `_p_estimated_size` propety didn't support deletion.
- Simplified implementation of the `_p_estimated_size` property to only accept integers. A TypeError is raised if an incorrect type is provided.

3.33 4.0.2 (2012-08-27)

- Correct initialization functions in renamed `_timestamp` extension.

3.34 4.0.1 (2012-08-26)

- Worked around test failure due to overflow to long on 32-bit systems.
- Renamed `TimeStamp` extension module to avoid clash with pure-Python `timestamp` module on case-insensitive filesystems.

  N.B: the canonical way to import the `TimeStamp` class is now:

```
from persistent.timestamp import TimeStamp
```

which will yield the class from the extension module (if available), falling back to the pure-Python reference implementation.

3.35 4.0.0 (2012-08-11)

3.35.1 Platform Changes

- Added explicit support for Python 3.2 and PyPy.
  - Note that the C implementations of Persistent, PickleCache, and Timestamp are not built (yet) on these platforms.
- Dropped support for Python < 2.6.

3.35.2 Testing Changes

- 100% unit test coverage.
- Removed all `ZODB`-dependent tests:
  - Rewrote some to avoid the dependency
  - Cloned the remainder into new `ZODB.tests` modules.
- Refactored some doctests refactored as unittests.
- Completed pure-Python reference implementations of 'Persistent', 'PickleCache', and 'TimeStamp'.
- All covered platforms tested under `tox`.
- Added support for continuous integration using `tox` and `jenkins`.
- Added `setup.py dev` alias (installs `nose` and `coverage`).
- Dropped dependency on `zope.testing` / `zope.testrunner`: tests now run with `setup.py test`.

3.35.3 Documentation Changes

- Refactored many Doctests as Sphinx documentation (snippets are exercised via 'tox').
- Added `setup.py docs` alias (installs `Sphinx` and `repoze.sphinx.autointerface`).

CHAPTER 4

Glossary

activation Moving an object from the `GHOST` state to the `UPTODATE` state, load its *pickled data* from its *jar*.

data manager The object responsible for storing and loading an object's *pickled data* in a backing store. Also called a *jar*.

deactivation Moving an object from the `UPTODATE` state to the `GHOST` state, discarding its *pickled data*.

ghost An object whose *pickled data* has not yet been loaded from its *jar*. Accessing or mutating any of its attributes causes that data to be loaded, which is referred to as *activation*.

invalidation Moving an object from either the `UPTODATE` state or the `CHANGED` state to the `GHOST` state, discarding its *pickled data*.

jar Alias for *data manager*: short for "pickle jar", because it traditionally holds the *pickled data* of persistent objects.

object cache An MRU cache for objects associated with a given *data manager*.

object id The stable identifier that uniquely names a particular object. This is analogous to Python's *id*, but unlike *id*, object ids remain the same for a given object across different processes.

pickled data The serialized data of a persistent object, stored in and retrieved from a backing store by a *data manager*.

volatile attribute Attributes of a persistent object which are *not* captured as part of its *pickled data*. These attributes thus disappear during *deactivation* or *invalidation*.

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

C

D

E

F

G

I

J

K

L

M

N

O

P

R

S

U

V