transaction Documentation

Release 1.2

Zope Foundation Contributors

Oct 23, 2018

Contents

A general transaction support library for Python.

The transaction package offers a two-phase commit protocol which allows multiple backends of any kind to participate in a transaction and commit their changes only if all of them can successfully do so. It also offers support for savepoints, so that part of a transaction can be rolled back without having to abort it completely.

There are already transaction backends for SQLAlchemy, ZODB, email, filesystem, and others. in addition, there are packages like pyramid_tm, which allows all the code in a web request to run inside of a transaction, and aborts the transaction automatically if an error occurs. It's also not difficult to create your own backends if necessary.

CHAPTER 1

Getting the transaction package

To install the transaction package you can use pip:

```
$ pip install transaction
```

After this, the package can be imported in your Python code, but there are a few things that we need to explain before doing that.

CHAPTER 2

Using transactions

At its simplest, the developer will use an existing transaction backend, and will at most require to commit or abort a transaction now and then. For example:

```
import transaction

try:
    # some code that uses one or more backends
    .
    .
    .
    transaction.commit()
except SomeError:
    transaction.abort()
```

CHAPTER 3

Things you need to know about the transaction machinery

3.1 Transactions

A transaction consists of one or more operations that we want to perform as a single action. It's an all or nothing proposition: either all the operations that are part of the transaction are completed successfully or none of them have any effect.

In the transaction package, a transaction object represents a running transaction that can be committed or aborted in the end.

3.2 Transaction managers

Applications interact with a transaction using a transaction manager, which is responsible for establishing the transaction boundaries. Basically this means that it creates the transactions and keeps track of the current one. Whenever an application wants to use the transaction machinery, it gets the current transaction from the transaction manager before starting any operations

The default transaction manager, *transaction.manager*, is thread local. You use it as a global variable, but every thread has it's own copy.[1]

Application developers will most likely never need to create their own transaction managers.

3.3 Data Managers

A data manager handles the interaction between the transaction manager and the data storage mechanism used by the application, which can be an object storage like the ZODB, a relational database, a file or any other storage mechanism that the application needs to control.

[1] The thread-local transaction manager, *transaction.manager* wraps a regular transaction manager. You can get the wrapped transaction manager using the *manager* attribute. Implementers of data managers can use this **advanced** feature to allow graceful shutdown from a central/main thread, by having their *close* methods call *unregisterSynch* on the wrapped transaction manager they obtained when created or opened.

The data manager provides a common interface for the transaction manager to use while a transaction is running. To be part of a specific transaction, a data manager has to 'join' it. Any number of data managers can join a transaction, which means that you could for example perform writing operations on a ZODB storage and a relational database as part of the same transaction. The transaction manager will make sure that both data managers can commit the transaction or none of them does.

An application developer will need to write a data manager for each different type of storage that the application uses. There are also third party data managers that can be used instead.

3.4 The two phase commit protocol

The transaction machinery uses a two phase commit protocol for coordinating all participating data managers in a transaction. The two phases work like follows:

1. The commit process is started.
2. Each associated data manager prepares the changes to be persistent.
3. Each data manager verifies that no errors or other exceptional conditions occurred during the attempt to persist the changes. If that happens, an exception should be raised. This is called 'voting'. A data manager votes 'no' by raising an exception if something goes wrong; otherwise, its vote is counted as a 'yes'.
4. If any of the associated data managers votes 'no', the transaction is aborted; otherwise, the changes are made permanent.

The two phase commit sequence requires that all the storages being used are capable of rolling back or aborting changes.

3.5 Savepoints

A savepoint allows a data manager to save work to its storage without committing the full transaction. In other words, the transaction will go on, but if a rollback is needed we can get back to this point instead of starting all over.

Savepoints are also useful to free memory that would otherwise be used to keep the whole state of the transaction. This can be very important when a transaction attempts a large number of changes.

CHAPTER 4

Additional Documentation

4.1 Using transactions with SQLAlchemy

Now that we got the terminology out of the way, let's show how to use this package in a Python application. One of the most popular ways of using the transaction package is to combine transactions from the ZODB with a relational database backend. Likewise, one of the most popular ways of communicating with a relational database in Python is to use the SQLAlchemy Object-Relational Mapper. Let's forget about the ZODB for the moment and show how one could use the transaction module in a Python application that needs to talk to a relational database.

4.1.1 Installing SQLAlchemy

Installing SQLAlchemy is as easy as installing any Python package available on PyPi:

```
$ pip install sqlalchemy
```

This will install the package in your Python environment. You'll need to set up a relational database that you can use to work out the examples in the following sections. SQLAlchemy supports most relational backends that you may have heard of, but the simplest thing to do is to use SQLite, since it doesn't require a separate Python driver. You'll have to make sure that the operating system packages required for using SQLite are present, though.

If you want to use another database, make sure you install the required system packages and drivers in addition to the database. For information about which databases are supported and where you can find the drivers, consult http://www.sqlalchemy.org/docs/core/engines.html#supported-dbapis.

4.1.2 Choosing a data manager

Hopefully, at this point SQLAlchemy and SQLite (or other database if you are feeling adventurous) are installed. To use this combination with the transaction package, we need a data manager that knows how to talk to SQLAlchemy so that the appropriate SQL commands are sent to SQLite whenever an event in the transaction life-cycle occurs.

Fortunately for us, there is already a package that does this on PyPI, so it's just a matter of installing it on our system. The package is called zope.sqlalchemy, but despite its name it doesn't depend on any zope packages other than zope.interface. By now you already know how to install it:

```
$ pip install zope.sqlalchemy
```

You can now create Python applications that use the transaction module to control any SQLAlchemy-supported relational backend.

4.1.3 A simple demonstration

It's time to show how to use SQLAlchemy together with the transaction package. To avoid lengthy digressions, knowledge of how SQLAlchemy works is assumed. If you are not familiar with that, reading the tutorial at http://www.sqlalchemy.org/docs/orm/tutorial.html will give you a good enough background to understand what follows.

After installing the required packages, you may wish to follow along the examples using the Python interpreter where you installed them. The first step is to create an engine:

```
1 >>> from sqlalchemy import create_engine
2 >>> engine = create_engine('sqlite:///:memory:')
```

This will connect us to the database. The connection string shown here is for SQLite, if you set up a different database you will need to look up the correct connection string syntax for it.

The next step is to define a class that will be mapped to a table in the relational database. SQLAlchemy's declarative syntax allows us to do that easily:

```
1  >>> from sqlalchemy import Column, Integer, String
2  >>> from sqlalchemy.ext.declarative import declarative_base
3  >>> Base = declarative_base()
4  >>> class User(Base):
5  >>>     __tablename__ = 'users'
6  ...
7  ...     id = Column(Integer, primary_key=True)
8  ...     name = Column(String)
9  ...     fullname = Column(String)
10 ...     password = Column(String)
11 ...
12 >>> Base.metadata.create_all(engine)
```

The User class is now mapped to the table named 'users'. The create_all method in line 12 creates the table in case it doesn't exist already.

We can now create a session and integrate the zope.sqlalchemy data manager with it so that we can use the transaction machinery. This is done by passing a Session Extension when creating the SQLAlchemy session:

```
1 >>> from sqlalchemy.orm import sessionmaker
2 >>> from zope.sqlalchemy import ZopeTransactionExtension
3 >>> Session = sessionmaker(bind=engine, extension=ZopeTransactionExtension())
4 >>> session = Session()
```

In line 3, we create a session class that is bound to the engine that we set up earlier. Notice how we pass the ZopeTransactionExtension using the extension parameter. This extension connects the SQLAlchemy session with the data manager provided by zope.sqlalchemy.

In line 4 we create a session. Under the hood, the ZopeTransactionExtension makes sure that the current transaction is joined by the zope.sqlalchemy data manager, so it's not necessary to explicitly join the transaction in our code.

Finally, we are able to put some data inside our new table and commit the transaction:

```
>>> import transaction
>>> session.add(User(id=1, name='John', fullname='John Smith', password='123'))
>>> transaction.commit()
```

Since the transaction was already joined by the zope.sqlalchemy data manager, we can just call commit and the transaction is correctly committed. As you can see, the integration between SQLAlchemy and the transaction machinery is pretty transparent.

4.1.4 Aborting transactions

Of course, when using the transaction machinery you can also abort or rollback a transaction. An example follows:

```
>>> session = Session()
>>> john = session.query(User).all()[0]
>>> john.fullname
u'John Smith'
>>> john.fullname = 'John Q. Public'
>>> john.fullname
u'John Q. Public'
>>> transaction.abort()
```

We need a new transaction for this example, so a new session is created. Since the old transaction had ended with the commit, creating a new session joins it to the current transaction, which will be a new one as well.

We make a query just to show that our user's fullname is 'John Smith', then we change that to 'John Q. Public'. When the transaction is aborted in line 8, the name is reverted to the old value.

If we create a new session and query the table for our old friend John, we'll see that the old value was indeed preserved because of the abort:

```
>>> session = Session()
>>> john = session.query(User).all()[0]
>>> john.fullname
u'John Smith'
```

4.1.5 Savepoints

A nice feature offered by many transactional backends is the existence of savepoints. These allow in effect to save the changes that we have made at the current point in a transaction, but without committing the transaction. If eventually we need to rollback a future operation, we can use the savepoint to return to the "safe" state that we had saved.

Unfortunately not every database supports savepoints and SQLite is precisely one of those that doesn't, which means that in order to be able to test this functionality you will have to install another database, like PostgreSQL. Of course, you can also just take our word that it really works, so suit yourself.

Let's see how a savepoint would work using PostgreSQL. First we'll import everything and setup the same table we used in our SQLite examples:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('postgresql://postgres@127.0.0.1:5432')
>>> from sqlalchemy import Column, Integer, String
>>> from sqlalchemy.ext.declarative import declarative_base
>>> Base = declarative_base()
```

(continues on next page)

(continued from previous page)

```
>>> Base.metadata.create_all(engine)
>>> class User(Base):
...     __tablename__ = 'users'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
>>> Base.metadata.create_all(engine)
>>> from sqlalchemy.orm import sessionmaker
>>> from zope.sqlalchemy import ZopeTransactionExtension
>>> Session = sessionmaker(bind=engine, extension=ZopeTransactionExtension())
```

We are now ready to create and use a savepoint:

```
>>> import transaction
>>> session = Session()
>>> session.add(User(id=1, name='John', fullname='John Smith', password='123'))
>>> sp = transaction.savepoint()
```

Everything should look familiar until line 4, where we create a savepoint and assign it to the sp variable. If we never need to rollback, this will not be used, but if course we have to hold on to it in case we do.

Now, we'll add a second user:

```
>>> session.add(User(id=2, name='John', fullname='John Watson', password='123'))
>>> [o.fullname for o in session.query(User).all()]
[u'John Smith', u'John Watson']
```

The new user has been added. We have not committed or aborted yet, but suppose we encounter an error condition that requires us to get rid of the new user, but not the one we added first. This is where the savepoint comes handy:

```
>>> sp.rollback()
>>> [o.fullname for o in session.query(User).all()]
[u'John Smith']
>>> transaction.commit()
```

As you can see, we just call the rollback method and we are back to where we wanted. The transaction can then be committed and the data that we decided to keep will be saved.

4.2 Managing more than one backend

Going through the previous section's examples, experienced users of any powerful enough relational backend might have been thinking, "wait, my database already can do that by itself. I can always commit or rollback when I want to, so what's the advantage of using this machinery?"

The answer is that if you are using a single backend and it already supports savepoints, you really don't need a transaction manager. The transaction machinery can still be useful with a single backend if it doesn't support transactions. A data manager can be written to add this support. There are existent packages that do this for files stored in a file system or for email sending, just to name a few examples.

However, the real power of the transaction manager is the ability to combine two or more of these data managers in a single transaction. Say you need to capture data from a form into a relational database and send email only on transaction commit, that's a good use case for the transaction package.

We will illustrate this by showing an example of coordinating transactions to a relational database and a ZODB client.

The first thing to do is set up the relational database, using the code that we've seen before:

```
>>> from sqlalchemy import create_engine
>>> engine = create_engine('postgresql://postgres@127.0.0.1:5432')
>>> from sqlalchemy import Column, Integer, String
>>> from sqlalchemy.ext.declarative import declarative_base
>>> Base = declarative_base()
>>> Base.metadata.create_all(engine)
>>> class User(Base):
...     __tablename__ = 'users'
...     id = Column(Integer, primary_key=True)
...     name = Column(String)
...     fullname = Column(String)
...     password = Column(String)
...
>>> Base.metadata.create_all(engine)
>>> from sqlalchemy.orm import sessionmaker
>>> from zope.sqlalchemy import ZopeTransactionExtension
>>> Session = sessionmaker(bind=engine, extension=ZopeTransactionExtension())
```

Now, let's set up a ZODB connection (you might need to install the ZODB first):

```
>>> from ZODB import DB, FileStorage

>>> storage = FileStorage.FileStorage('test.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()
```

We're ready for adding a user to the relational database table. Right after that, we add some data to the ZODB using the user name as key:

```
>>> import transaction
>>> session.add(User(id=1, name='John', fullname='John Smith', password='123'))
>>> root['John'] = 'some data that goes into the object database'
```

Since both the ZopeTransactionExtension and the ZODB connection join the transaction automatically, we can just make the changes we want and be ready to commit the transaction immediately.

```
>>> transaction.commit()
```

Again, both the SQLAlchemy and the ZODB data managers joined the transaction, so that we can commit the transaction and both backends save the data. If there's a problem with one of the backends, the transaction is aborted in both regardless of the state of the other. It's also possible to abort the transaction manually, of course, causing a rollback on both backends as well.

4.3 Transaction convenience support

(We *really* need to write proper documentation for the transaction package, but I don't want to block the conveniences documented here for that.)

4.3.1 with support

We can now use the with statement to define transaction boundaries.

```
>>> import transaction.tests.savepointsample
>>> dm = transaction.tests.savepointsample.SampleSavepointDataManager()
>>> list(dm.keys())
[]
```

We can use it with a manager:

```
>>> with transaction.manager as t:
...     dm['z'] = 3
...     t.note(u'test 3')

>>> dm['z']
3

>>> dm.last_note == u'test 3'
True

>>> with transaction.manager: #doctest ELLIPSIS
...     dm['z'] = 4
...     xxx
Traceback (most recent call last):
...
NameError: ... name 'xxx' is not defined

>>> dm['z']
3
```

On Python 2, you can also abbreviate `with transaction.manager:` as `with transaction:`. This does not work on Python 3 (see see http://bugs.python.org/issue12022).

4.3.2 Retries

Commits can fail for transient reasons, especially conflicts. Applications will often retry transactions some number of times to overcome transient failures. This typically looks something like:

```
for i in range(3):
    try:
       with transaction.manager:
           ... some something ...
    except SomeTransientException:
       continue
    else:
       break
```

This is rather ugly and easy to get wrong.

Transaction managers provide two helpers for this case.

Running and retrying functions as transactions

The first helper runs a function as a transaction:

```
def do_somthing():
    "Do something"
    ... some something ...
```

(continues on next page)

(continued from previous page)

```
transaction.manager.run(do_somthing)
```

You can also use this as a decorator, which executes the decorated function immediately[1]:

```
@transaction.manager.run
def _():
    "Do something"
    ... some something ...
```

The transaction manager `run` method will run the function and return the results. If the function raises a `TransientError`, the function will be retried a configurable number of times, 3 by default. Any other exceptions will be raised.

The function name (if it isn't `'_'`) and docstring, if any, are added to the transaction description.

You can pass an integer number of times to try to the `run` method:

```
transaction.manager.run(do_somthing, 9)

@transaction.manager.run(9)
def _():
    "Do something"
    ... some something ...
```

The default number of times to try is 3.

Retrying code blocks using a attempt iterator

An older helper for running transactions uses an iterator of attempts:

```
for attempt in transaction.manager.attempts():
    with attempt as t:
        ... some something ...
```

This runs the code block until it runs without a transient error or until the number of attempts is exceeded. By default, it tries 3 times, but you can pass a number of attempts:

```
for attempt in transaction.manager.attempts(9):
    with attempt as t:
        ... some something ...
```

4.4 Dooming Transactions

A doomed transaction behaves exactly the same way as an active transaction but raises an error on any attempt to commit it, thus forcing an abort.

Doom is useful in places where abort is unsafe and an exception cannot be raised. This occurs when the programmer wants the code following the doom to run but not commit. It is unsafe to abort in these circumstances as a following get() may implicitly open a new transaction.

Any attempt to commit a doomed transaction will raise a DoomedTransaction exception.

[1] Some people find this easier to read, even though the result isn't a decorated function, but rather the result of calling it in a transaction. The function name _ is used here to emphasize that the function is essentially being used as an anonymous function.

An example of such a use case can be found in zope/app/form/browser/editview.py. Here a form validation failure must doom the transaction as committing the transaction may have side-effects. However, the form code must continue to calculate a form containing the error messages to return.

For Zope in general, code running within a request should always doom transactions rather than aborting them. It is the responsibilty of the publication to either abort() or commit() the transaction. Application code can use savepoints and doom() safely.

To see how it works we first need to create a stub data manager:

```
>>> from transaction.interfaces import IDataManager
>>> from zope.interface import implementer
>>> @implementer(IDataManager)
... class DataManager:
...     def __init__(self):
...         self.attr_counter = {}
...     def __getattr__(self, name):
...         def f(transaction):
...             self.attr_counter[name] = self.attr_counter.get(name, 0) + 1
...         return f
...     def total(self):
...         count = 0
...         for access_count in self.attr_counter.values():
...             count += access_count
...         return count
...     def sortKey(self):
...         return 1
```

Start a new transaction:

```
>>> import transaction
>>> txn = transaction.begin()
>>> dm = DataManager()
>>> txn.join(dm)
```

We can ask a transaction if it is doomed to avoid expensive operations. An example of a use case is an object-relational mapper where a pre-commit hook sends all outstanding SQL to a relational database for objects changed during the transaction. This expensive operation is not necessary if the transaction has been doomed. A non-doomed transaction should return False:

```
>>> txn.isDoomed()
False
```

We can doom a transaction by calling .doom() on it:

```
>>> txn.doom()
>>> txn.isDoomed()
True
```

We can doom it again if we like:

```
>>> txn.doom()
```

The data manager is unchanged at this point:

```
>>> dm.total()
0
```

Attempting to commit a doomed transaction any number of times raises a DoomedTransaction:

```
>>> txn.commit()
Traceback (most recent call last):
DoomedTransaction: transaction doomed, cannot commit
>>> txn.commit()
Traceback (most recent call last):
DoomedTransaction: transaction doomed, cannot commit
```

But still leaves the data manager unchanged:

```
>>> dm.total()
0
```

But the doomed transaction can be aborted:

```
>>> txn.abort()
```

Which aborts the data manager:

```
>>> dm.total()
1
>>> dm.attr_counter['abort']
1
```

Dooming the current transaction can also be done directly from the transaction module. We can also begin a new transaction directly after dooming the old one:

```
>>> txn = transaction.begin()
>>> transaction.isDoomed()
False
>>> transaction.doom()
>>> transaction.isDoomed()
True
>>> txn = transaction.begin()
```

After committing a transaction we get an assertion error if we try to doom the transaction. This could be made more specific, but trying to doom a transaction after it's been committed is probably a programming error:

```
>>> txn = transaction.begin()
>>> txn.commit()
>>> txn.doom()
Traceback (most recent call last):
    ...
ValueError: non-doomable
```

A doomed transaction should act the same as an active transaction, so we should be able to join it:

```
>>> txn = transaction.begin()
>>> txn.doom()
>>> dm2 = DataManager()
>>> txn.join(dm2)
```

Clean up:

```
>>> txn = transaction.begin()
>>> txn.abort()
```

4.5 Savepoints

Savepoints provide a way to save to disk intermediate work done during a transaction allowing:

- partial transaction (subtransaction) rollback (abort)
- state of saved objects to be freed, freeing on-line memory for other uses

Savepoints make it possible to write atomic subroutines that don't make top-level transaction commitments.

4.5.1 Applications

To demonstrate how savepoints work with transactions, we've provided a sample data manager implementation that provides savepoint support. The primary purpose of this data manager is to provide code that can be read to understand how savepoints work. The secondary purpose is to provide support for demonstrating the correct operation of savepoint support within the transaction system. This data manager is very simple. It provides flat storage of named immutable values, like strings and numbers.

```
>>> import transaction
>>> from transaction.tests import savepointsample
>>> dm = savepointsample.SampleSavepointDataManager()
>>> dm['name'] = 'bob'
```

As with other data managers, we can commit changes:

```
>>> transaction.commit()
>>> dm['name']
'bob'
```

and abort changes:

```
>>> dm['name'] = 'sally'
>>> dm['name']
'sally'
>>> transaction.abort()
>>> dm['name']
'bob'
```

Now, let's look at an application that manages funds for people. It allows deposits and debits to be entered for multiple people. It accepts a sequence of entries and generates a sequence of status messages. For each entry, it applies the change and then validates the user's account. If the user's account is invalid, we roll back the change for that entry. The success or failure of an entry is indicated in the output status. First we'll initialize some accounts:

```
>>> dm['bob-balance'] = 0.0
>>> dm['bob-credit'] = 0.0
>>> dm['sally-balance'] = 0.0
>>> dm['sally-credit'] = 100.0
>>> transaction.commit()
```

Now, we'll define a validation function to validate an account:

```
>>> def validate_account(name):
...     if dm[name+'-balance'] + dm[name+'-credit'] < 0:
...         raise ValueError('Overdrawn', name)
```

And a function to apply entries. If the function fails in some unexpected way, it rolls back all of its changes and prints the error:

```
>>> def apply_entries(entries):
...     savepoint = transaction.savepoint()
...     try:
...         for name, amount in entries:
...             entry_savepoint = transaction.savepoint()
...             try:
...                 dm[name+'-balance'] += amount
...                 validate_account(name)
...             except ValueError as error:
...                 entry_savepoint.rollback()
...                 print("%s %s" % ('Error', str(error)))
...             else:
...                 print("%s %s" % ('Updated', name))
...     except Exception as error:
...         savepoint.rollback()
...         print("%s" % ('Unexpected exception'))
```

Now let's try applying some entries:

```
>>> apply_entries([
...     ('bob',   10.0),
...     ('sally', 10.0),
...     ('bob',   20.0),
...     ('sally', 10.0),
...     ('bob',   -100.0),
...     ('sally', -100.0),
...     ])
Updated bob
Updated sally
Updated bob
Updated sally
Error ('Overdrawn', 'bob')
Updated sally

>>> dm['bob-balance']
30.0

>>> dm['sally-balance']
-80.0
```

If we provide entries that cause an unexpected error:

```
>>> apply_entries([
...     ('bob',   10.0),
...     ('sally', 10.0),
...     ('bob',   '20.0'),
...     ('sally', 10.0),
...     ])
Updated bob
Updated sally
Unexpected exception
```

Because the apply_entries used a savepoint for the entire function, it was able to rollback the partial changes without rolling back changes made in the previous call to `apply_entries`:

```
>>> dm['bob-balance']
30.0
```

(continues on next page)

(continued from previous page)

```
>>> dm['sally-balance']
-80.0
```

If we now abort the outer transactions, the earlier changes will go away:

```
>>> transaction.abort()

>>> dm['bob-balance']
0.0

>>> dm['sally-balance']
0.0
```

4.5.2 Savepoint invalidation

A savepoint can be used any number of times:

```
>>> dm['bob-balance'] = 100.0
>>> dm['bob-balance']
100.0
>>> savepoint = transaction.savepoint()

>>> dm['bob-balance'] = 200.0
>>> dm['bob-balance']
200.0
>>> savepoint.rollback()
>>> dm['bob-balance']
100.0

>>> savepoint.rollback()  # redundant, but should be harmless
>>> dm['bob-balance']
100.0

>>> dm['bob-balance'] = 300.0
>>> dm['bob-balance']
300.0
>>> savepoint.rollback()
>>> dm['bob-balance']
100.0
```

However, using a savepoint invalidates any savepoints that come after it:

```
>>> dm['bob-balance'] = 200.0
>>> dm['bob-balance']
200.0
>>> savepoint1 = transaction.savepoint()

>>> dm['bob-balance'] = 300.0
>>> dm['bob-balance']
300.0
>>> savepoint2 = transaction.savepoint()

>>> savepoint.rollback()
>>> dm['bob-balance']
```

(continues on next page)

(continued from previous page)

```
100.0

>>> savepoint2.rollback()
Traceback (most recent call last):
...
InvalidSavepointRollbackError: invalidated by a later savepoint

>>> savepoint1.rollback()
Traceback (most recent call last):
...
InvalidSavepointRollbackError: invalidated by a later savepoint

>>> transaction.abort()
```

4.5.3 Databases without savepoint support

Normally it's an error to use savepoints with databases that don't support savepoints:

```
>>> dm_no_sp = savepointsample.SampleDataManager()
>>> dm_no_sp['name'] = 'bob'
>>> transaction.commit()
>>> dm_no_sp['name'] = 'sally'
>>> transaction.savepoint()
Traceback (most recent call last):
...
TypeError: ('Savepoints unsupported', {'name': 'bob'})

>>> transaction.abort()
```

However, a flag can be passed to the transaction savepoint method to indicate that databases without savepoint support should be tolerated until a savepoint is rolled back. This allows transactions to proceed if there are no reasons to roll back:

```
>>> dm_no_sp['name'] = 'sally'
>>> savepoint = transaction.savepoint(1)
>>> dm_no_sp['name'] = 'sue'
>>> transaction.commit()
>>> dm_no_sp['name']
'sue'

>>> dm_no_sp['name'] = 'sam'
>>> savepoint = transaction.savepoint(1)
>>> savepoint.rollback()
Traceback (most recent call last):
...
TypeError: ('Savepoints unsupported', {'name': 'sam'})
```

4.5.4 Failures

If a failure occurs when creating or rolling back a savepoint, the transaction state will be uncertain and the transaction will become uncommitable. From that point on, most transaction operations, including commit, will fail until the transaction is aborted.

In the previous example, we got an error when we tried to rollback the savepoint. If we try to commit the transaction, the commit will fail:

```
>>> transaction.commit()
Traceback (most recent call last):
...
TransactionFailedError: An operation previously failed, with traceback:
...
TypeError: ('Savepoints unsupported', {'name': 'sam'})
```

We have to abort it to make any progress:

```
>>> transaction.abort()
```

Similarly, in our earlier example, where we tried to take a savepoint with a data manager that didn't support savepoints:

```
>>> dm_no_sp['name'] = 'sally'
>>> dm['name'] = 'sally'
>>> savepoint = transaction.savepoint()
Traceback (most recent call last):
...
TypeError: ('Savepoints unsupported', {'name': 'sue'})

>>> transaction.commit()
Traceback (most recent call last):
...
TransactionFailedError: An operation previously failed, with traceback:
...
TypeError: ('Savepoints unsupported', {'name': 'sue'})


>>> transaction.abort()
```

After clearing the transaction with an abort, we can get on with new transactions:

```
>>> dm_no_sp['name'] = 'sally'
>>> dm['name'] = 'sally'
>>> transaction.commit()
>>> dm_no_sp['name']
'sally'
>>> dm['name']
'sally'
```

4.6 Hooking the Transaction Machinery

The `transaction` machinery allows application developers to register two different groups of callbacks to be called, one group before committing the transaction and one group after.

These hooks are **not** designed to be used as replacements for the two-phase commit machinery defined by a resource manager (see *Writing a Resource Manager*). In particular, hook functions **must not** raise or propagate exceptions.

> **Warning:** Hook functions which *do* raise or propagate exceptions will leave the application in an undefined state.

4.6.1 The `addBeforeCommitHook()` Method

Let's define a hook to call, and a way to see that it was called.

```
>>> log = []
>>> def reset_log():
...     del log[:]

>>> def hook(arg='no_arg', kw1='no_kw1', kw2='no_kw2'):
...     log.append("arg %r kw1 %r kw2 %r" % (arg, kw1, kw2))
```

Now register the hook with a transaction.

```
>>> from transaction import begin
>>> import transaction
>>> t = begin()
>>> t.addBeforeCommitHook(hook, '1')
```

We can see that the hook is indeed registered.

```
>>> [(hook.__name__, args, kws)
...  for hook, args, kws in t.getBeforeCommitHooks()]
[('hook', ('1',), {})]
```

When transaction commit starts, the hook is called, with its arguments.

```
>>> log
[]
>>> t.commit()
>>> log
["arg '1' kw1 'no_kw1' kw2 'no_kw2'"]
>>> reset_log()
```

A hook's registration is consumed whenever the hook is called. Since the hook above was called, it's no longer registered:

```
>>> from transaction import commit
>>> len(list(t.getBeforeCommitHooks()))
0
>>> commit()
>>> log
[]
```

The hook is only called for a full commit, not for a savepoint.

```
>>> t = begin()
>>> t.addBeforeCommitHook(hook, 'A', dict(kw1='B'))
>>> dummy = t.savepoint()
>>> log
[]
>>> t.commit()
>>> log
["arg 'A' kw1 'B' kw2 'no_kw2'"]
>>> reset_log()
```

If a transaction is aborted, no hook is called.

```
>>> from transaction import abort
>>> t = begin()
>>> t.addBeforeCommitHook(hook, ["OOPS!"])
>>> abort()
>>> log
[]
>>> commit()
>>> log
[]
```

The hook is called before the commit does anything, so even if the commit fails the hook will have been called. To provoke failures in commit, we'll add failing resource manager to the transaction.

```
>>> class CommitFailure(Exception):
...     pass
>>> class FailingDataManager:
...     def tpc_begin(self, txn, sub=False):
...         raise CommitFailure('failed')
...     def abort(self, txn):
...         pass

>>> t = begin()
>>> t.join(FailingDataManager())

>>> t.addBeforeCommitHook(hook, '2')

>>> from transaction.tests.common import DummyFile
>>> from transaction.tests.common import Monkey
>>> from transaction.tests.common import assertRaisesEx
>>> from transaction import _transaction
>>> buffer = DummyFile()
>>> with Monkey(_transaction, _TB_BUFFER=buffer):
...     err = assertRaisesEx(CommitFailure, t.commit)
>>> log
["arg '2' kw1 'no_kw1' kw2 'no_kw2'"]
>>> reset_log()
```

Let's register several hooks.

```
>>> t = begin()
>>> t.addBeforeCommitHook(hook, '4', dict(kw1='4.1'))
>>> t.addBeforeCommitHook(hook, '5', dict(kw2='5.2'))
```

They are returned in the same order by getBeforeCommitHooks.

```
>>> [(hook.__name__, args, kws)
...  for hook, args, kws in t.getBeforeCommitHooks()]
[('hook', ('4',), {'kw1': '4.1'}),
('hook', ('5',), {'kw2': '5.2'})]
```

And commit also calls them in this order.

```
>>> t.commit()
>>> len(log)
2
>>> log
["arg '4' kw1 '4.1' kw2 'no_kw2'",
```

(continues on next page)

(continued from previous page)

```
"arg '5' kw1 'no_kw1' kw2 '5.2'"]
>>> reset_log()
```

While executing, a hook can itself add more hooks, and they will all be called before the real commit starts.

```
>>> def recurse(txn, arg):
...     log.append('rec' + str(arg))
...     if arg:
...         txn.addBeforeCommitHook(hook, '-')
...         txn.addBeforeCommitHook(recurse, (txn, arg-1))

>>> t = begin()
>>> t.addBeforeCommitHook(recurse, (t, 3))
>>> commit()
>>> log
['rec3',
        "arg '-' kw1 'no_kw1' kw2 'no_kw2'",
'rec2',
        "arg '-' kw1 'no_kw1' kw2 'no_kw2'",
'rec1',
        "arg '-' kw1 'no_kw1' kw2 'no_kw2'",
'rec0']
>>> reset_log()
```

4.6.2 The `addAfterCommitHook()` Method

Let's define a hook to call, and a way to see that it was called.

```
>>> log = []
>>> def reset_log():
...     del log[:]

>>> def hook(status, arg='no_arg', kw1='no_kw1', kw2='no_kw2'):
...     log.append("%r arg %r kw1 %r kw2 %r" % (status, arg, kw1, kw2))
```

Now register the hook with a transaction.

```
>>> from transaction import begin
>>> t = begin()
>>> t.addAfterCommitHook(hook, '1')
```

We can see that the hook is indeed registered.

```
>>> [(hook.__name__, args, kws)
...  for hook, args, kws in t.getAfterCommitHooks()]
[('hook', ('1',), {})]
```

When transaction commit is done, the hook is called, with its arguments.

```
>>> log
[]
>>> t.commit()
>>> log
["True arg '1' kw1 'no_kw1' kw2 'no_kw2'"]
>>> reset_log()
```

A hook's registration is consumed whenever the hook is called. Since the hook above was called, it's no longer registered:

```
>>> from transaction import commit
>>> len(list(t.getAfterCommitHooks()))
0
>>> commit()
>>> log
[]
```

The hook is only called after a full commit, not for a savepoint.

```
>>> t = begin()
>>> t.addAfterCommitHook(hook, 'A', dict(kw1='B'))
>>> dummy = t.savepoint()
>>> log
[]
>>> t.commit()
>>> log
["True arg 'A' kw1 'B' kw2 'no_kw2'"]
>>> reset_log()
```

If a transaction is aborted, no hook is called.

```
>>> from transaction import abort
>>> t = begin()
>>> t.addAfterCommitHook(hook, ["OOPS!"])
>>> abort()
>>> log
[]
>>> commit()
>>> log
[]
```

The hook is called after the commit is done, so even if the commit fails the hook will have been called. To provoke failures in commit, we'll add failing resource manager to the transaction.

```
>>> class CommitFailure(Exception):
...     pass
>>> class FailingDataManager:
...     def tpc_begin(self, txn):
...         raise CommitFailure('failed')
...     def abort(self, txn):
...         pass

>>> t = begin()
>>> t.join(FailingDataManager())

>>> t.addAfterCommitHook(hook, '2')
>>> from transaction.tests.common import DummyFile
>>> from transaction.tests.common import Monkey
>>> from transaction.tests.common import assertRaisesEx
>>> from transaction import _transaction
>>> buffer = DummyFile()
>>> with Monkey(_transaction, _TB_BUFFER=buffer):
...     err = assertRaisesEx(CommitFailure, t.commit)
>>> log
```

(continues on next page)

(continued from previous page)

```
["False arg '2' kw1 'no_kw1' kw2 'no_kw2'"]
>>> reset_log()
```

Let's register several hooks.

```
>>> t = begin()
>>> t.addAfterCommitHook(hook, '4', dict(kw1='4.1'))
>>> t.addAfterCommitHook(hook, '5', dict(kw2='5.2'))
```

They are returned in the same order by getAfterCommitHooks.

```
>>> [(hook.__name__, args, kws)
...  for hook, args, kws in t.getAfterCommitHooks()]
[('hook', ('4',), {'kw1': '4.1'}),
('hook', ('5',), {'kw2': '5.2'})]
```

And commit also calls them in this order.

```
>>> t.commit()
>>> len(log)
2
>>> log
["True arg '4' kw1 '4.1' kw2 'no_kw2'",
"True arg '5' kw1 'no_kw1' kw2 '5.2'"]
>>> reset_log()
```

While executing, a hook can itself add more hooks, and they will all be called before the real commit starts.

```
>>> def recurse(status, txn, arg):
...     log.append('rec' + str(arg))
...     if arg:
...         txn.addAfterCommitHook(hook, '-')
...         txn.addAfterCommitHook(recurse, (txn, arg-1))

>>> t = begin()
>>> t.addAfterCommitHook(recurse, (t, 3))
>>> commit()
>>> log
['rec3',
        "True arg '-' kw1 'no_kw1' kw2 'no_kw2'",
'rec2',
        "True arg '-' kw1 'no_kw1' kw2 'no_kw2'",
'rec1',
        "True arg '-' kw1 'no_kw1' kw2 'no_kw2'",
'rec0']
>>> reset_log()
```

If an after commit hook is raising an exception then it will log a message at error level so that if other hooks are registered they can be executed. We don't support execution dependencies at this level.

```
>>> from transaction import TransactionManager
>>> from transaction.tests.test_manager import DataObject
>>> mgr = TransactionManager()
>>> do = DataObject(mgr)

>>> def hookRaise(status, arg='no_arg', kw1='no_kw1', kw2='no_kw2'):
```

(continues on next page)

(continued from previous page)

```
...     raise TypeError("Fake raise")

>>> t = begin()

>>> t.addAfterCommitHook(hook, ('-', 1))
>>> t.addAfterCommitHook(hookRaise, ('-', 2))
>>> t.addAfterCommitHook(hook, ('-', 3))
>>> commit()

>>> log
["True arg '-' kw1 1 kw2 'no_kw2'", "True arg '-' kw1 3 kw2 'no_kw2'"]

>>> reset_log()
```

Test that the associated transaction manager has been cleaned up when after commit hooks are registered

```
>>> mgr = TransactionManager()
>>> do = DataObject(mgr)

>>> t = begin()
>>> t._manager._txn is not None
True

>>> t.addAfterCommitHook(hook, ('-', 1))
>>> commit()

>>> log
["True arg '-' kw1 1 kw2 'no_kw2'"]

>>> t._manager._txn is not None
False

>>> reset_log()
```

4.7 Writing a Data Manager

4.7.1 Simple Data Manager

```
>>> from transaction.tests.examples import DataManager
```

This `transaction.tests.examples.DataManager` class provides a trivial data-manager implementation and docstrings to illustrate the the protocol and to provide a tool for writing tests.

Our sample data manager has state that is updated through an inc method and through transaction operations.

When we create a sample data manager:

```
>>> dm = DataManager()
```

It has two bits of state, state:

```
>>> dm.state
0
```

and delta:

```
>>> dm.delta
0
```

Both of which are initialized to 0. state is meant to model committed state, while delta represents tentative changes within a transaction. We change the state by calling inc:

```
>>> dm.inc()
```

which updates delta:

```
>>> dm.delta
1
```

but state isn't changed until we commit the transaction:

```
>>> dm.state
0
```

To commit the changes, we use 2-phase commit. We execute the first stage by calling prepare. We need to pass a transation. Our sample data managers don't really use the transactions for much, so we'll be lazy and use strings for transactions:

```
>>> t1 = '1'
>>> dm.prepare(t1)
```

The sample data manager updates the state when we call prepare:

```
>>> dm.state
1
>>> dm.delta
1
```

This is mainly so we can detect some affect of calling the methods.

Now if we call commit:

```
>>> dm.commit(t1)
```

Our changes are"permanent". The state reflects the changes and the delta has been reset to 0.

```
>>> dm.state
1
>>> dm.delta
0
```

4.7.2 The `prepare()` Method

Prepare to commit data

```
>>> dm = DataManager()
>>> dm.inc()
>>> t1 = '1'
>>> dm.prepare(t1)
>>> dm.commit(t1)
```

(continues on next page)

(continued from previous page)

```
>>> dm.state
1
>>> dm.inc()
>>> t2 = '2'
>>> dm.prepare(t2)
>>> dm.abort(t2)
>>> dm.state
1
```

It is en error to call prepare more than once without an intervening commit or abort:

```
>>> dm.prepare(t1)

>>> dm.prepare(t1)
Traceback (most recent call last):
...
TypeError: Already prepared

>>> dm.prepare(t2)
Traceback (most recent call last):
...
TypeError: Already prepared

>>> dm.abort(t1)
```

If there was a preceeding savepoint, the transaction must match:

```
>>> rollback = dm.savepoint(t1)
>>> dm.prepare(t2)
Traceback (most recent call last):
,,,
TypeError: ('Transaction missmatch', '2', '1')

>>> dm.prepare(t1)
```

4.7.3 The `abort()` method

The abort method can be called before two-phase commit to throw away work done in the transaction:

```
>>> dm = DataManager()
>>> dm.inc()
>>> dm.state, dm.delta
(0, 1)
>>> t1 = '1'
>>> dm.abort(t1)
>>> dm.state, dm.delta
(0, 0)
```

The abort method also throws away work done in savepoints:

```
>>> dm.inc()
>>> r = dm.savepoint(t1)
>>> dm.inc()
>>> r = dm.savepoint(t1)
>>> dm.state, dm.delta
```

(continues on next page)

(continued from previous page)

```
(0, 2)
>>> dm.abort(t1)
>>> dm.state, dm.delta
(0, 0)
```

If savepoints are used, abort must be passed the same transaction:

```
>>> dm.inc()
>>> r = dm.savepoint(t1)
>>> t2 = '2'
>>> dm.abort(t2)
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '1')

>>> dm.abort(t1)
```

The abort method is also used to abort a two-phase commit:

```
>>> dm.inc()
>>> dm.state, dm.delta
(0, 1)
>>> dm.prepare(t1)
>>> dm.state, dm.delta
(1, 1)
>>> dm.abort(t1)
>>> dm.state, dm.delta
(0, 0)
```

Of course, the transactions passed to prepare and abort must match:

```
>>> dm.prepare(t1)
>>> dm.abort(t2)
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '1')

>>> dm.abort(t1)
```

4.7.4 The `commit()` method

Called to omplete two-phase commit

```
>>> dm = DataManager()
>>> dm.state
0
>>> dm.inc()
```

We start two-phase commit by calling prepare:

```
>>> t1 = '1'
>>> dm.prepare(t1)

We complete it by calling commit:
```

```
>>> dm.commit(t1)
>>> dm.state
1
```

It is an error ro call commit without calling prepare first:

```
>>> dm.inc()
>>> t2 = '2'
>>> dm.commit(t2)
Traceback (most recent call last):
...
TypeError: Not prepared to commit

>>> dm.prepare(t2)
>>> dm.commit(t2)
```

If course, the transactions given to prepare and commit must be the same:

```
>>> dm.inc()
>>> t3 = '3'
>>> dm.prepare(t3)
>>> dm.commit(t2)
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '3')
```

4.7.5 The `savepoint()` method

Provide the ability to rollback transaction state

Savepoints provide a way to:

- Save partial transaction work. For some data managers, this could allow resources to be used more efficiently.
- Provide the ability to revert state to a point in a transaction without aborting the entire transaction. In other words, savepoints support partial aborts.

Savepoints don't use two-phase commit. If there are errors in setting or rolling back to savepoints, the application should abort the containing transaction. This is *not* the responsibility of the data manager.

Savepoints are always associated with a transaction. Any work done in a savepoint's transaction is tentative until the transaction is committed using two-phase commit.

```
>>> dm = DataManager()
>>> dm.inc()
>>> t1 = '1'
>>> r = dm.savepoint(t1)
>>> dm.state, dm.delta
(0, 1)
>>> dm.inc()
>>> dm.state, dm.delta
(0, 2)
>>> r.rollback()
>>> dm.state, dm.delta
(0, 1)
>>> dm.prepare(t1)
>>> dm.commit(t1)
```

(continues on next page)

(continued from previous page)

```
>>> dm.state, dm.delta
(1, 0)
```

Savepoints must have the same transaction:

```
>>> r1 = dm.savepoint(t1)
>>> dm.state, dm.delta
(1, 0)
>>> dm.inc()
>>> dm.state, dm.delta
(1, 1)
>>> t2 = '2'
>>> r2 = dm.savepoint(t2)
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '1')

>>> r2 = dm.savepoint(t1)
>>> dm.inc()
>>> dm.state, dm.delta
(1, 2)
```

If we rollback to an earlier savepoint, we discard all work done later:

```
>>> r1.rollback()
>>> dm.state, dm.delta
(1, 0)
```

and we can no longer rollback to the later savepoint:

```
>>> r2.rollback()
Traceback (most recent call last):
...
TypeError: ('Attempt to roll back to invalid save point', 3, 2)
```

We can roll back to a savepoint as often as we like:

```
>>> r1.rollback()
>>> r1.rollback()
>>> r1.rollback()
>>> dm.state, dm.delta
(1, 0)

>>> dm.inc()
>>> dm.inc()
>>> dm.inc()
>>> dm.state, dm.delta
(1, 3)
>>> r1.rollback()
>>> dm.state, dm.delta
(1, 0)
```

But we can't rollback to a savepoint after it has been committed:

```
>>> dm.prepare(t1)
>>> dm.commit(t1)
```

(continues on next page)

(continued from previous page)

```
>>> r1.rollback()
Traceback (most recent call last):
...
TypeError: Attempt to rollback stale rollback
```

4.8 Writing a Resource Manager

4.8.1 Simple Resource Manager

```
>>> from transaction.tests.examples import ResourceManager
```

This `transaction.tests.examples.ResourceManager` class provides a trivial resource-manager implementation and doc strings to illustrate the protocol and to provide a tool for writing tests.

Our sample resource manager has state that is updated through an inc method and through transaction operations.

When we create a sample resource manager:

```
>>> rm = ResourceManager()
```

It has two pieces state, state and delta, both initialized to 0:

```
>>> rm.state
0
>>> rm.delta
0
```

state is meant to model committed state, while delta represents tentative changes within a transaction. We change the state by calling inc:

```
>>> rm.inc()
```

which updates delta:

```
>>> rm.delta
1
```

but state isn't changed until we commit the transaction:

```
>>> rm.state
0
```

To commit the changes, we use 2-phase commit. We execute the first stage by calling prepare. We need to pass a transation. Our sample resource managers don't really use the transactions for much, so we'll be lazy and use strings for transactions. The sample resource manager updates the state when we call tpc_vote:

```
>>> t1 = '1'
>>> rm.tpc_begin(t1)
>>> rm.state, rm.delta
(0, 1)

>>> rm.tpc_vote(t1)
```

(continues on next page)

(continued from previous page)

```
>>> rm.state, rm.delta
(1, 1)

Now if we call tpc_finish:

>>> rm.tpc_finish(t1)
```

Our changes are "permanent". The state reflects the changes and the delta has been reset to 0.

```
>>> rm.state, rm.delta
(1, 0)
```

4.8.2 The `tpc_begin()` Method

Called by the transaction manager to ask the RM to prepare to commit data.

```
>>> rm = ResourceManager()
>>> rm.inc()
>>> t1 = '1'
>>> rm.tpc_begin(t1)
>>> rm.tpc_vote(t1)
>>> rm.tpc_finish(t1)
>>> rm.state
1
>>> rm.inc()
>>> t2 = '2'
>>> rm.tpc_begin(t2)
>>> rm.tpc_vote(t2)
>>> rm.tpc_abort(t2)
>>> rm.state
1
```

It is an error to call tpc_begin more than once without completing two-phase commit:

```
>>> rm.tpc_begin(t1)

>>> rm.tpc_begin(t1)
Traceback (most recent call last):
...
ValueError: txn in state 'tpc_begin' but expected one of (None,)
>>> rm.tpc_abort(t1)
```

If there was a preceeding savepoint, the transaction must match:

```
>>> rollback = rm.savepoint(t1)
>>> rm.tpc_begin(t2)
Traceback (most recent call last):
,,,
TypeError: ('Transaction missmatch', '2', '1')

>>> rm.tpc_begin(t1)
```

4.8.3 The `tpc_vote()` Method

Verify that a data manager can commit the transaction.

This is the last chance for a data manager to vote 'no'. A data manager votes 'no' by raising an exception.

Passed *transaction*, which is the ITransaction instance associated with the transaction being committed.

4.8.4 The `tpc_finish()` Method

Complete two-phase commit

```
>>> rm = ResourceManager()
>>> rm.state
0
>>> rm.inc()

We start two-phase commit by calling prepare:

>>> t1 = '1'
>>> rm.tpc_begin(t1)
>>> rm.tpc_vote(t1)

We complete it by calling tpc_finish:

>>> rm.tpc_finish(t1)
>>> rm.state
1
```

It is an error ro call tpc_finish without calling tpc_vote:

```
>>> rm.inc()
>>> t2 = '2'
>>> rm.tpc_begin(t2)
>>> rm.tpc_finish(t2)
Traceback (most recent call last):
...
ValueError: txn in state 'tpc_begin' but expected one of ('tpc_vote',)

>>> rm.tpc_abort(t2)  # clean slate

>>> rm.tpc_begin(t2)
>>> rm.tpc_vote(t2)
>>> rm.tpc_finish(t2)
```

Of course, the transactions given to tpc_begin and tpc_finish must be the same:

```
>>> rm.inc()
>>> t3 = '3'
>>> rm.tpc_begin(t3)
>>> rm.tpc_vote(t3)
>>> rm.tpc_finish(t2)
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '3')
```

4.8.5 The `tpc_abort()` Method

Abort a transaction

The abort method can be called before two-phase commit to throw away work done in the transaction:

```
>>> rm = ResourceManager()
>>> rm.inc()
>>> rm.state, rm.delta
(0, 1)
>>> t1 = '1'
>>> rm.tpc_abort(t1)
>>> rm.state, rm.delta
(0, 0)
```

The abort method also throws away work done in savepoints:

```
>>> rm.inc()
>>> r = rm.savepoint(t1)
>>> rm.inc()
>>> r = rm.savepoint(t1)
>>> rm.state, rm.delta
(0, 2)
>>> rm.tpc_abort(t1)
>>> rm.state, rm.delta
(0, 0)
```

If savepoints are used, abort must be passed the same transaction:

```
>>> rm.inc()
>>> r = rm.savepoint(t1)
>>> t2 = '2'
>>> rm.tpc_abort(t2)
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '1')

>>> rm.tpc_abort(t1)
```

The abort method is also used to abort a two-phase commit:

```
>>> rm.inc()
>>> rm.state, rm.delta
(0, 1)
>>> rm.tpc_begin(t1)
>>> rm.state, rm.delta
(0, 1)
>>> rm.tpc_vote(t1)
>>> rm.state, rm.delta
(1, 1)
>>> rm.tpc_abort(t1)
>>> rm.state, rm.delta
(0, 0)
```

Of course, the transactions passed to prepare and abort must match:

```
>>> rm.tpc_begin(t1)
>>> rm.tpc_abort(t2)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '1')

>>> rm.tpc_abort(t1)
```

This should never fail.

4.8.6 The `savepoint()` Method

Provide the ability to rollback transaction state

Savepoints provide a way to:

- Save partial transaction work. For some resource managers, this could allow resources to be used more efficiently.
- Provide the ability to revert state to a point in a transaction without aborting the entire transaction. In other words, savepoints support partial aborts.

Savepoints don't use two-phase commit. If there are errors in setting or rolling back to savepoints, the application should abort the containing transaction. This is *not* the responsibility of the resource manager.

Savepoints are always associated with a transaction. Any work done in a savepoint's transaction is tentative until the transaction is committed using two-phase commit.

```
>>> rm = ResourceManager()
>>> rm.inc()
>>> t1 = '1'
>>> r = rm.savepoint(t1)
>>> rm.state, rm.delta
(0, 1)
>>> rm.inc()
>>> rm.state, rm.delta
(0, 2)
>>> r.rollback()
>>> rm.state, rm.delta
(0, 1)
>>> rm.tpc_begin(t1)
>>> rm.tpc_vote(t1)
>>> rm.tpc_finish(t1)
>>> rm.state, rm.delta
(1, 0)
```

Savepoints must have the same transaction:

```
>>> r1 = rm.savepoint(t1)
>>> rm.state, rm.delta
(1, 0)
>>> rm.inc()
>>> rm.state, rm.delta
(1, 1)
>>> t2 = '2'
>>> r2 = rm.savepoint(t2)
Traceback (most recent call last):
...
TypeError: ('Transaction missmatch', '2', '1')
```

(continues on next page)

(continued from previous page)

```
>>> r2 = rm.savepoint(t1)
>>> rm.inc()
>>> rm.state, rm.delta
(1, 2)
```

If we rollback to an earlier savepoint, we discard all work done later:

```
>>> r1.rollback()
>>> rm.state, rm.delta
(1, 0)
```

and we can no longer rollback to the later savepoint:

```
>>> r2.rollback()
Traceback (most recent call last):
...
TypeError: ('Attempt to roll back to invalid save point', 3, 2)
```

We can roll back to a savepoint as often as we like:

```
>>> r1.rollback()
>>> r1.rollback()
>>> r1.rollback()
>>> rm.state, rm.delta
(1, 0)

>>> rm.inc()
>>> rm.inc()
>>> rm.inc()
>>> rm.state, rm.delta
(1, 3)
>>> r1.rollback()
>>> rm.state, rm.delta
(1, 0)
```

But we can't rollback to a savepoint after it has been committed:

```
>>> rm.tpc_begin(t1)
>>> rm.tpc_vote(t1)
>>> rm.tpc_finish(t1)

>>> r1.rollback()
Traceback (most recent call last):
...
TypeError: Attempt to rollback stale rollback
```

4.9 Transaction integrations / Data Manager Implementations

The following packages have been integrated with the `transaction` package so that their transactions can be integerated with others.

ZODB ZODB was the original user of the `transaction` package. Its transactions are controlled by `transaction` and ZODB fully implements the 2-phase commit protocol.

SQLAlchemy An Object Relational Mapper for Python, SQLAlchemy can use zope.sqlalchemy to have its transactions integrated with others.

repoze.sendmail repoze.sendmail allows coupling the sending of email messages with a transaction, using the Zope transaction manager. This allows messages to only be sent out when and if a transaction is committed, preventing users from receiving notifications about events which may not have completed successfully.

4.10 `transaction` API Reference

4.10.1 Interfaces

interface `transaction.interfaces.`**ITransactionManager**

An object that manages a sequence of transactions.

Applications use transaction managers to establish transaction boundaries.

explicit

Explicit mode indicator.

This is true if the transaction manager is in explicit mode. In explicit mode, transactions must be begun explicitly, by calling `begin()` and ended explicitly by calling `commit()` or `abort()`.

begin()

Explicitly begin and return a new transaction.

If an existing transaction is in progress and the transaction manager not in explicit mode, the previous transaction will be aborted. If an existing transaction is in progress and the transaction manager is in explicit mode, an `AlreadyInTransaction` exception will be raised..

The `newTransaction` method of registered synchronizers is called, passing the new transaction object.

Note that when not in explicit mode, transactions may be started implicitly without calling `begin`. In that case, `newTransaction` isn't called because the transaction manager doesn't know when to call it. The transaction is likely to have begun long before the transaction manager is involved. (Conceivably the `commit` and `abort` methods could call `begin`, but they don't.)

get()

Get the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

commit()

Commit the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

abort()

Abort the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

doom()

Doom the current transaction.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

isDoomed()

Returns True if the current transaction is doomed, otherwise False.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

savepoint(*optimistic=False*)
Create a savepoint from the current transaction.

If the optimistic argument is true, then data managers that don't support savepoints can be used, but an error will be raised if the savepoint is rolled back.

An ISavepoint object is returned.

In explicit mode, if a transaction hasn't begun, a `NoTransaction` exception will be raised.

registerSynch(*synch*)
Register an ISynchronizer.

Synchronizers are notified about some major events in a transaction's life. See ISynchronizer for details.

If a synchronizer registers while there is an active transaction, its newTransaction method will be called with the active transaction.

unregisterSynch(*synch*)
Unregister an ISynchronizer.

Synchronizers are notified about some major events in a transaction's life. See ISynchronizer for details.

clearSynchs()
Unregister all registered ISynchronizers.

This exists to support test cleanup/initialization

registeredSynchs()
Determine if any ISynchronizers are registered.

Return true if any are registered, and return False otherwise.

This exists to support test cleanup/initialization

interface `transaction.interfaces.`**ITransaction**
Object representing a running transaction.

Objects with this interface may represent different transactions during their lifetime (.begin() can be called to start a new transaction using the same instance, although that example is deprecated and will go away in ZODB 3.6).

user
A user name associated with the transaction.

The format of the user name is defined by the application. The value is text (unicode). Storages record the user value, as meta-data, when a transaction commits.

A storage may impose a limit on the size of the value; behavior is undefined if such a limit is exceeded (for example, a storage may raise an exception, or truncate the value).

description
A textual description of the transaction.

The value is text (unicode). Method note() is the intended way to set the value. Storages record the description, as meta-data, when a transaction commits.

A storage may impose a limit on the size of the description; behavior is undefined if such a limit is exceeded (for example, a storage may raise an exception, or truncate the value).

extension
A dictionary containing application-defined metadata.

commit()
Finalize the transaction.

This executes the two-phase commit algorithm for all IDataManager objects associated with the transaction.

abort()
Abort the transaction.

This is called from the application. This can only be called before the two-phase commit protocol has been started.

doom()
Doom the transaction.

Dooms the current transaction. This will cause DoomedTransactionException to be raised on any attempt to commit the transaction.

Otherwise the transaction will behave as if it was active.

savepoint(*optimistic=False*)
Create a savepoint.

If the optimistic argument is true, then data managers that don't support savepoints can be used, but an error will be raised if the savepoint is rolled back.

An ISavepoint object is returned.

join(*datamanager*)
Add a data manager to the transaction.

datamanager must provide the transactions.interfaces.IDataManager interface.

note(*text*)
Add text (unicode) to the transaction description.

This modifies the *.description* attribute; see its docs for more detail. First surrounding whitespace is stripped from *text*. If *.description* is currently an empty string, then the stripped text becomes its value, else two newlines and the stripped text are appended to *.description*.

setExtendedInfo(*name*, *value*)
Add extension data to the transaction.

name is the text (unicode) name of the extension property to set

value must be picklable and json serializable (not an instance).

Multiple calls may be made to set multiple extension properties, provided the names are distinct.

Storages record the extension data, as meta-data, when a transaction commits.

A storage may impose a limit on the size of extension data; behavior is undefined if such a limit is exceeded (for example, a storage may raise an exception, or remove <*name, value*> pairs).

addBeforeCommitHook(*hook*, *args=()*, *kws=None*)
Register a hook to call before the transaction is committed.

The specified hook function will be called after the transaction's commit method has been called, but before the commit process has been started. The hook will be passed the specified positional (*args*) and keyword (*kws*) arguments. *args* is a sequence of positional arguments to be passed, defaulting to an empty tuple (no positional arguments are passed). *kws* is a dictionary of keyword argument names and values to be passed, or the default None (no keyword arguments are passed).

Multiple hooks can be registered and will be called in the order they were registered (first registered, first called). This method can also be called from a hook: an executing hook can register more hooks. Applications should take care to avoid creating infinite loops by recursively registering hooks.

Hooks are called only for a top-level commit. A savepoint creation does not call any hooks. If the transaction is aborted, hooks are not called, and are discarded. Calling a hook "consumes" its registration too: hook registrations do not persist across transactions. If it's desired to call the same hook on every transaction commit, then addBeforeCommitHook() must be called with that hook during every transaction; in such a case consider registering a synchronizer object via a TransactionManager's registerSynch() method instead.

getBeforeCommitHooks()
Return iterable producing the registered addBeforeCommit hooks.

A triple (hook, args, kws) is produced for each registered hook. The hooks are produced in the order in which they would be invoked by a top-level transaction commit.

addAfterCommitHook(*hook*, *args=()*, *kws=None*)
Register a hook to call after a transaction commit attempt.

The specified hook function will be called after the transaction commit succeeds or aborts. The first argument passed to the hook is a Boolean value, true if the commit succeeded, or false if the commit aborted. *args* specifies additional positional, and *kws* keyword, arguments to pass to the hook. *args* is a sequence of positional arguments to be passed, defaulting to an empty tuple (only the true/false success argument is passed). *kws* is a dictionary of keyword argument names and values to be passed, or the default None (no keyword arguments are passed).

Multiple hooks can be registered and will be called in the order they were registered (first registered, first called). This method can also be called from a hook: an executing hook can register more hooks. Applications should take care to avoid creating infinite loops by recursively registering hooks.

Hooks are called only for a top-level commit. A savepoint creation does not call any hooks. Calling a hook "consumes" its registration: hook registrations do not persist across transactions. If it's desired to call the same hook on every transaction commit, then addAfterCommitHook() must be called with that hook during every transaction; in such a case consider registering a synchronizer object via a TransactionManager's registerSynch() method instead.

getAfterCommitHooks()
Return iterable producing the registered addAfterCommit hooks.

A triple (hook, args, kws) is produced for each registered hook. The hooks are produced in the order in which they would be invoked by a top-level transaction commit.

set_data(*ob*, *data*)
Hold data on behalf of an object

For objects such as data managers or their subobjects that work with multiple transactions, it's convenient to store transaction-specific data on the transaction itself. The transaction knows nothing about the data, but simply holds it on behalf of the object.

The object passed should be the object that needs the data, as opposed to simple object like a string. (Internally, the id of the object is used as the key.)

data(*ob*)
Retrieve data held on behalf of an object.

See set_data.

isRetryableError(*error*)
Determine if the error is retryable.

Return true if any joined IRetryDataManager considers the error transient. Such errors may occur due to concurrency issues in the underlying storage engine.

interface transaction.interfaces.**IDataManager**
Objects that manage transactional storage.

These objects may manage data for other objects, or they may manage non-object storages, such as relational databases. For example, a ZODB.Connection.

Note that when some data is modified, that data's data manager should join a transaction so that data can be committed when the user commits the transaction.

transaction_manager
The transaction manager (TM) used by this data manager.

This is a public attribute, intended for read-only use. The value is an instance of ITransactionManager, typically set by the data manager's constructor.

abort(*transaction*)
Abort a transaction and forget all changes.

Abort must be called outside of a two-phase commit.

Abort is called by the transaction manager to abort transactions that are not yet in a two-phase commit. It may also be called when rolling back a savepoint made before the data manager joined the transaction.

In any case, after abort is called, the data manager is no longer participating in the transaction. If there are new changes, the data manager must rejoin the transaction.

tpc_begin(*transaction*)
Begin commit of a transaction, starting the two-phase commit.

transaction is the ITransaction instance associated with the transaction being committed.

commit(*transaction*)
Commit modifications to registered objects.

Save changes to be made persistent if the transaction commits (if tpc_finish is called later). If tpc_abort is called later, changes must not persist.

This includes conflict detection and handling. If no conflicts or errors occur, the data manager should be prepared to make the changes persist when tpc_finish is called.

tpc_vote(*transaction*)
Verify that a data manager can commit the transaction.

This is the last chance for a data manager to vote 'no'. A data manager votes 'no' by raising an exception.

transaction is the ITransaction instance associated with the transaction being committed.

tpc_finish(*transaction*)
Indicate confirmation that the transaction is done.

Make all changes to objects modified by this transaction persist.

transaction is the ITransaction instance associated with the transaction being committed.

This should never fail. If this raises an exception, the database is not expected to maintain consistency; it's a serious error.

tpc_abort(*transaction*)
Abort a transaction.

This is called by a transaction manager to end a two-phase commit on the data manager. Abandon all changes to objects modified by this transaction.

transaction is the ITransaction instance associated with the transaction being committed.

This should never fail.

sortKey()
Return a key to use for ordering registered DataManagers.

In order to guarantee a total ordering, keys must be strings.

ZODB uses a global sort order to prevent deadlock when it commits transactions involving multiple resource managers. The resource manager must define a sortKey() method that provides a global ordering for resource managers.

interface transaction.interfaces.**ISavepointDataManager**
Extends: *transaction.interfaces.IDataManager*

savepoint()
Return a data-manager savepoint (IDataManagerSavepoint).

interface transaction.interfaces.**IRetryDataManager**
Extends: *transaction.interfaces.IDataManager*

should_retry(*exception*)
Return whether a given exception instance should be retried.

A data manager can provide this method to indicate that a a transaction that raised the given error should be retried. This method may be called by an ITransactionManager when considering whether to retry a failed transaction.

interface transaction.interfaces.**IDataManagerSavepoint**
Savepoint for data-manager changes for use in transaction savepoints.

Datamanager savepoints are used by, and only by, transaction savepoints.

Note that data manager savepoints don't have any notion of, or responsibility for, validity. It isn't the responsibility of data-manager savepoints to prevent multiple rollbacks or rollbacks after transaction termination. Preventing invalid savepoint rollback is the responsibility of transaction rollbacks. Application code should never use data-manager savepoints.

rollback()
Rollback any work done since the savepoint.

interface transaction.interfaces.**ISavepoint**
A transaction savepoint.

rollback()
Rollback any work done since the savepoint.

InvalidSavepointRollbackError is raised if the savepoint isn't valid.

valid
Boolean indicating whether the savepoint is valid

class transaction.interfaces.**InvalidSavepointRollbackError**
Attempt to rollback an invalid savepoint.

A savepoint may be invalid because:

- The surrounding transaction has committed or aborted.
- An earlier savepoint in the same transaction has been rolled back.

interface transaction.interfaces.**ISynchronizer**
Objects that participate in the transaction-boundary notification API.

beforeCompletion(*transaction*)
Hook that is called by the transaction at the start of a commit.

afterCompletion(*transaction*)
Hook that is called by the transaction after completing a commit.

newTransaction(*transaction*)
Hook that is called at the start of a transaction.

This hook is called when, and only when, a transaction manager's begin() method is called explictly.

class transaction.interfaces.**TransactionError**
An error occurred due to normal transaction processing.

class transaction.interfaces.**TransactionFailedError**
Cannot perform an operation on a transaction that previously failed.

An attempt was made to commit a transaction, or to join a transaction, but this transaction previously raised an exception during an attempt to commit it. The transaction must be explicitly aborted, either by invoking abort() on the transaction, or begin() on its transaction manager.

class transaction.interfaces.**DoomedTransaction**
A commit was attempted on a transaction that was doomed.

class transaction.interfaces.**TransientError**
An error has occured when performing a transaction.

It's possible that retrying the transaction will succeed.

4.10.2 API Objects

class transaction._transaction.**Transaction**(*synchronizers=None*, *manager=None*)

isDoomed()
See ITransaction.

doom()
See ITransaction.

join(*resource*)
See ITransaction.

savepoint(*optimistic=False*)
See ITransaction.

register(*obj*)
See ITransaction.

commit()
See ITransaction.

getBeforeCommitHooks()
See ITransaction.

addBeforeCommitHook(*hook*, *args=()*, *kws=None*)
See ITransaction.

getAfterCommitHooks()
See ITransaction.

addAfterCommitHook(*hook*, *args=()*, *kws=None*)
See ITransaction.

abort()
See ITransaction.

note(*text*)
See ITransaction.

setUser(*user_name*, *path=u'/'*)
See ITransaction.

setExtendedInfo(*name*, *value*)
See ITransaction.

class transaction._transaction.**Savepoint**(*transaction*, *optimistic*, **resources*)
Transaction savepoint.

Transaction savepoints coordinate savepoints for data managers participating in a transaction.

rollback()
See ISavepoint.

class transaction._manager.**TransactionManager**(*explicit=False*)

__enter__()
Alias for *get()*

__exit__(*t*, *v*, *tb*)
On error, aborts the current transaction. Otherwise, commits.

begin()
See ITransactionManager.

get()
See ITransactionManager.

registerSynch(*synch*)
See ITransactionManager.

unregisterSynch(*synch*)
See ITransactionManager.

clearSynchs()
See ITransactionManager.

registeredSynchs()
See ITransactionManager.

isDoomed()
See ITransactionManager.

doom()
See ITransactionManager.

commit()
See ITransactionManager.

abort()
See ITransactionManager.

savepoint(*optimistic=False*)
See ITransactionManager.

class transaction._manager.**ThreadTransactionManager**
Thread-local transaction manager.

A thread-local transaction manager can be used as a global variable, but has a separate copy for each thread.

Advanced applications can use the *manager* attribute to get a wrapped TransactionManager to allow cross-thread calls for graceful shutdown of data managers.

4.11 `transaction` Developer Documentation

Transaction objects manage resources for an individual activity.

4.11.1 Compatibility issues

The implementation of Transaction objects involves two layers of backwards compatibility, because this version of transaction supports both ZODB 3 and ZODB 4. Zope is evolving towards the ZODB4 interfaces.

Transaction has two methods for a resource manager to call to participate in a transaction – register() and join(). join() takes a resource manager and adds it to the list of resources. register() is for backwards compatibility. It takes a persistent object and registers its _p_jar attribute. TODO: explain adapter

4.11.2 Two-phase commit

A transaction commit involves an interaction between the transaction object and one or more resource managers. The transaction manager calls the following four methods on each resource manager; it calls tpc_begin() on each resource manager before calling commit() on any of them.

1. tpc_begin(txn)
2. commit(txn)
3. tpc_vote(txn)
4. tpc_finish(txn)

4.11.3 Before-commit hook

Sometimes, applications want to execute some code when a transaction is committed. For example, one might want to delay object indexing until a transaction commits, rather than indexing every time an object is changed. Or someone might want to check invariants only after a set of operations. A pre-commit hook is available for such use cases: use addBeforeCommitHook(), passing it a callable and arguments. The callable will be called with its arguments at the start of the commit (but not for substransaction commits).

4.11.4 After-commit hook

Sometimes, applications want to execute code after a transaction commit attempt succeeds or aborts. For example, one might want to launch non transactional code after a successful commit. Or still someone might want to launch asynchronous code after. A post-commit hook is available for such use cases: use addAfterCommitHook(), passing it a callable and arguments. The callable will be called with a Boolean value representing the status of the commit operation as first argument (true if successfull or false iff aborted) preceding its arguments at the start of the commit (but not for substransaction commits). Commit hooks are not called for transaction.abort().

4.11.5 Error handling

When errors occur during two-phase commit, the transaction manager aborts all the resource managers. The specific methods it calls depend on whether the error occurs before or after the call to tpc_vote() on that transaction manager.

If the resource manager has not voted, then the resource manager will have one or more uncommitted objects. There are two cases that lead to this state; either the transaction manager has not called commit() for any objects on this resource manager or the call that failed was a commit() for one of the objects of this resource manager. For each uncommitted object, including the object that failed in its commit(), call abort().

Once uncommitted objects are aborted, tpc_abort() or abort_sub() is called on each resource manager.

4.11.6 Synchronization

You can register sychronization objects (synchronizers) with the tranasction manager. The synchronizer must implement beforeCompletion() and afterCompletion() methods. The transaction manager calls beforeCompletion() when it starts a top-level two-phase commit. It calls afterCompletion() when a top-level transaction is committed or aborted. The methods are passed the current Transaction as their only argument.

4.11.7 Explicit vs implicit transactions

By default, transactions are implicitly managed. Calling `begin()` on a transaction manager implicitly aborts the previous transaction and calling `commit()` or `abort()` implicitly begins a new one. This behavior can be convenient for interactive use, but invites subtle bugs:

- Calling begin() without realizing that there are outstanding changes that will be aborted.
- Interacting with a database without controlling transactions, in which case changes may be unexpectedly discarded.

For applications, including frameworks that control transactions, transaction managers provide an optional explicit mode. Transaction managers have an `explicit` constructor keyword argument that, if True puts the transaction manager in explicit mode. In explicit mode:

- It is an error to call `get()`, `commit()`, `abort()`, `doom()`, `isDoomed`, or `savepoint()` without a preceding `begin()` call. Doing so will raise a `NoTransaction` exception.
- It is an error to call `begin()` after a previous `begin()` without an intervening `commit()` or `abort()` call. Doing so will raise an `AlreadyInTransaction` exception.

In explicit mode, bugs like those mentioned above are much easier to avoid because they cause explicit exceptions that can typically be caught in development.

An additional benefit of explicit mode is that it can allow data managers to manage resources more efficiently.

Transaction managers have an explicit attribute that can be queried to determine if explicit mode is enabled.

Python Module Index

t

Index

Symbols

A

B

C

D

E

G

U

V