

---

# **transaction Documentation**

*Release 3.1.0.dev0*

**Zope Foundation Contributors**

**Aug 04, 2022**



---

## Contents

---

<b>1</b>	<b>Changes</b>	<b>3</b>
<b>2</b>	<b>Transaction convenience support</b>	<b>11</b>
<b>3</b>	<b>Dooming Transactions</b>	<b>15</b>
<b>4</b>	<b>Savepoints</b>	<b>19</b>
<b>5</b>	<b>Hooking the Transaction Machinery</b>	<b>25</b>
<b>6</b>	<b>Writing a Data Manager</b>	<b>33</b>
<b>7</b>	<b>Transaction integrations / Data Manager Implementations</b>	<b>41</b>
<b>8</b>	<b>Using transactions with SQLAlchemy</b>	<b>43</b>
<b>9</b>	<b>Managing more than one backend</b>	<b>47</b>
<b>10</b>	<b>transaction API Reference</b>	<b>49</b>
<b>11</b>	<b>transaction Developer Notes</b>	<b>61</b>
<b>12</b>	<b>Getting the transaction package</b>	<b>65</b>
<b>13</b>	<b>Using transactions</b>	<b>67</b>
<b>14</b>	<b>Things you need to know about the transaction machinery</b>	<b>69</b>
	<b>Python Module Index</b>	<b>71</b>
	<b>Index</b>	<b>73</b>



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.

## **Additional Documentation**



### 1.1 3.1.0 (unreleased)

- Add support for Python 3.9, 3.10.

### 1.2 3.0.1 (2020-12-11)

- Exception raised by a before commit hook is no longer hidden. No further commit hooks are called and exception is propagated to the caller of `commit()`. See [#95](#).

### 1.3 3.0.0 (2019-12-11)

- Drop support for Python 3.4.
- Add support for Python 3.8.
- Drop support for legacy transaction APIs including `Transaction.register()` and old ZODB3-style data-managers. See [issue 89](#).
- `TransactionManager.run` now commits/aborts the transaction “active” after the execution of *func* (and no longer the initial transaction which might already have been committed/aborted by *func*) ([#58](#)).

It aborts the transaction now for all exceptions raised by *func* - even if it is only an instance of `BaseException` but not of `Exception`, such as e.g. a `SystemExit` or `KeyboardInterrupt` exception.

- Support abort hooks (symmetrically to commit hooks) ([#77](#)).
- Make `Transaction` drop references to its hooks, manager, synchronizers and data after a successful `commit()` and after *any* `abort()`. This helps avoid potential cyclic references. See [issue 82](#).
- Allow synchronizers to access `Transaction.data()` when their `afterCompletion` method is called while aborting a transaction.

- Make it safe to call `Transaction.abort()` more than once. The second and subsequent calls are no-ops. Previously a `ValueError(Foreign transaction)` would be raised.

## 1.4 2.4.0 (2018-10-23)

- Changed the implementation of `ThreadTransactionManager` to be a `thread.local` that wraps a `TransactionManager` rather than a `thread.local` that inherits from `TransactionManager`. It now exposes a `manager` attribute that allows access to the wrapped transaction manager to allow cross thread calls. See [issue 68](#).

## 1.5 2.3.0 (2018-10-19)

- Add support for Python 3.7.
- Reach 100% test coverage.
- Fix `transaction.manager.run` formatting transaction notes when given a mix of byte and text strings, such as can happen in Python 2 with `unicode_literals`.

## 1.6 2.2.1 (2018-03-27)

- Make documentation index more user friendly; move old docs to developer section.
- Don't crash when printing tracebacks in IPython on Python 2. (This addresses <https://github.com/zopefoundation/transaction/issues/5>.)

## 1.7 2.2.0 (2018-02-27)

- Add support for Python 3.6.
- Drop support for Python 3.3.
- Add `isRetryableError` to the `transaction.interfaces.ITransaction` interface to allow external systems to query whether an exception is retryable (transient) by any of the attached data managers. Any `transaction.interfaces.TransientError` is considered retryable but a data manager may also consider other exceptions on a per-instance basis.

See <https://github.com/zopefoundation/transaction/pull/38>

## 1.8 2.1.2 (2017-03-11)

- To avoid leaking memory, don't include unexpected value in warnings about non-text transaction meta data.

## 1.9 2.1.1 (2017-03-11)

- For backward compatibility, relax the requirements that transaction meta data (user or description) be text:
  - If `None` is assigned, the assignment is ignored.



- If a non-text value is assigned, a warning is issued and the value is converted to text. If the value is a binary string, it will be decoded with the UTF-8 encoding the `replace` error policy.

## 1.10 2.1.0 (2017-02-08)

Added a transaction-manager explicit mode. Explicit mode makes some kinds of application bugs easier to detect and potentially allows data managers to manage resources more efficiently.

(This addresses <https://github.com/zopefoundation/transaction/issues/35>.)

## 1.11 2.0.3 (2016-11-17)

- The user and description fields must now be set with text (unicode) data. Previously, if bytes were provided, they'd be decoded as ASCII. It was decided that this would lead to bugs that were hard to test for.

Also, the transaction meta-data field, `extended_info` has been renamed to `extension`.

## 1.12 2.0.2 (2016-11-13)

- Fixed: Some legacy applications expect the transaction `_extension` attribute to be mutable and it wasn't.

## 1.13 2.0.1 (2016-11-11)

- The transaction `user` and `description` attributes are now defined to be text (unicode) as opposed to Python the `str` type.
- Added the `extended_info` transaction attribute which contains transaction meta data. (The `_extension` attribute is retained as an alias for backward compatibility.)

The transaction interface, `ITransaction`, now requires `extended_info` keys to be text (unicode) and values to be JSON-serializable.

- Removed `setUser` from `ITransaction`. We'll keep the method indefinitely, but it's unseemly in `ITransaction`. :)

The main purpose of these changes is to tighten up the text specification of `user`, `description` and `extended_info` keys, and to give us more flexibility in the future for serializing extended info. It's possible that these changes will be breaking, so we're also increasing the major version number.

## 1.14 1.7.0 (2016-11-08)

- Added a transaction-manager `run` method for running a function as a transaction, retrying as necessary on transient errors.
- Fixed the transaction manager `attempts` method. It didn't stop repeating when there wasn't an error.
- Corrected `ITransaction` by removing `beforeCommitHook` (which is no longer implemented) and removing 'self' from two methods.

## 1.15 1.6.1 (2016-06-10)

- Fixed: Synchronizers that registered with transaction managers when transactions were in progress didn't have their `newTransaction` methods called to let them know of the in-progress transactions.

## 1.16 1.6.0 (2016-05-21)

- New transaction API for storing data on behalf of objects, such as data managers.
- Drop references to data managers joined to a transaction when it is committed or aborted.

## 1.17 1.5.0 (2016-05-05)

- Drop support for Python 2.6 and 3.2.
- Add support for Python 3.5.
- Added APIs for interrogating and clearing internal state to support client tests.

## 1.18 1.4.4 (2015-05-19)

- Use the standard `valuerefs()` method rather than relying on implementation details of `WeakValueDictionary` in `WeakSet`.
- Add support for PyPy3.
- Require 100% branch coverage (in addition to 100% statement coverage).

## 1.19 1.4.3 (2014-03-20)

- Add support for Python 3.4.

## 1.20 1.4.2 (skipped)

- Released in error as 1.4.3.

## 1.21 1.4.1 (2013-02-20)

- Document that values returned by `sortKey` must be strings, in order to guarantee total ordering.
- Fix occasional `RuntimeError: dictionary changed size during iteration` errors in `transaction.weakset` on Python 3.

## 1.22 1.4.0 (2013-01-03)

- Updated Trove classifiers.

## 1.23 1.4.0b1 (2012-12-18)

- Converted existing doctests into Sphinx documentation (snippets are exercised via ‘tox’).
- 100% unit test coverage.
- Backward incompatibility: raise `ValueError` rather than `AssertionError` for runtime errors:
  - In `Transaction.doom` if the transaction is in a non-doomable state.
  - In `TransactionManager.attempts` if passed a non-positive value.
  - In `TransactionManager.free` if passed a foreign transaction.
- Declared support for Python 3.3 in `setup.py`, and added `tox` testing.
- When a non-retryable exception was raised as the result of a call to `transaction.manager.commit` within the “attempts” machinery, the exception was not reraised properly. Symptom: an unrecoverable exception such as `Unsupported: Storing blobs in <somestorage> is not supported.` would be swallowed inappropriately.

## 1.24 1.3.0 (2012-05-16)

- Added Sphinx API documentation.
- Added explicit support for PyPy.
- Dropped use of Python3-incompatible `zope.interface.implements` class advisor in favor of `zope.interface.implementer` class decorator.
- Added support for continuous integration using `tox` and `jenkins`.
- Added `setup.py docs` alias (installs Sphinx and dependencies).
- Added `setup.py dev` alias (runs `setup.py develop` plus installs `nose` and `coverage`).
- Python 3.3 compatibility.
- Fix “for attempt in `transaction.attempts(x)`” machinery, which would not retry a transaction if its implicit call to `.commit()` itself raised a transient error. Symptom: seeing conflict errors even though you thought you were retrying some number of times via the “attempts” machinery (the first attempt to generate an exception during commit would cause that exception to be raised).

## 1.25 1.2.0 (2011-12-05)

New Features:

- Python 3.2 compatibility.
- Dropped Python 2.4 and 2.5 compatibility (use 1.1.1 if you need to use “transaction” under these Python versions).

## 1.26 1.1.1 (2010-09-16)

### Bug Fixes:

- Code in `_transaction.py` held on to local references to traceback objects after calling `sys.exc_info()` to get one, causing potential reference leakages.
- Fixed `hexlify NameError` in `transaction._transaction.oid_repr` and add test.

## 1.27 1.1.0 (1010-05-12)

### New Features:

- Transaction managers and the transaction module can be used with the `with` statement to define transaction boundaries, as in:

```
with transaction:
    ... do some things ...
```

See `transaction/tests/convenience.txt` for more details.

- There is a new iterator function that automates dealing with transient errors (such as ZODB conflict errors). For example, in:

```
for attempt in transaction.attempts(5):
    with attempt:
        ... do some things ..
```

If the work being done raises transient errors, the transaction will be retried up to 5 times.

See `transaction/tests/convenience.txt` for more details.

### Bugs fixed:

- Fixed a bug that caused extra commit calls to be made on data managers under certain special circumstances.  
<https://mail.zope.org/pipermail/zodb-dev/2010-May/013329.html>
- When threads were reused, transaction data could leak accross them, causing subtle application bugs.  
<https://bugs.launchpad.net/zodb/+bug/239086>

## 1.28 1.0.1 (2010-05-07)

- LP #142464: remove double newline between log entries: it makes doing smarter formatting harder.
- Updated tests to remove use of deprecated `zope.testing.doctest`.

## 1.29 1.0.0 (2009-07-24)

- Fix test that incorrectly relied on the order of a list that was generated from a dict.
- Remove crufty `DEPENDENCIES.cfg` left over from `zpkg`.

### 1.30 1.0a1 (2007-12-18)

- Initial release, branched from ZODB trunk on 2007-11-08 (aka “3.9.0dev”).
- Remove (deprecated) support for beforeCommitHook alias to addBeforeCommitHook.
- Add weakset tests.
- Remove unit tests that depend on ZODB.tests.utils from test\_transaction (these are actually integration tests).



---

## Transaction convenience support

---

### 2.1 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 == '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 <http://bugs.python.org/issue12022>).

## 2.2 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.

### 2.2.1 Running and retrying functions as transactions

The first helper runs a function as a transaction:

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

transaction.manager.run(do_something)
```

You can also use this as a decorator, which executes the decorated function immediately<sup>1</sup>:

```
@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_something, 9)

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

The default number of times to try is 3.

---

<sup>1</sup> 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.



## 2.2.2 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 ...
```



---

## 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.

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 responsibility 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()
```



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.

### 4.1 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()
```

(continues on next page)

(continued from previous page)

```
>>> 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
```

(continues on next page)



(continued from previous page)

```
>>> 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

>>> 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
```

## 4.2 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
```

(continues on next page)

(continued from previous page)

```
>>> 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']
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()
```

## 4.3 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'})
```

## 4.4 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'
```

---

## 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 *resourcemanager*). 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.

### 5.1 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',))
```

(continues on next page)

(continued from previous page)

```

>>> 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'",
 "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()

```

## 5.2 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
["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'):
...     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

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

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

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

(continues on next page)

(continued from previous page)

```
>>> t._manager is None
True
>>> mgr._txn is None
True

>>> reset_log()
```



---

## Writing a Data Manager

---

### 6.1 Simple Data Manager

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

This class provides a trivial *IDataManager* implementation and doc strings to illustrate 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:

```
>>> rm = DataManager()
```

It has two pieces, `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 `tpc_begin`. We need to pass a transaction. Our sample data managers don't really use the transactions for much, so we'll be lazy and use strings for transactions. The sample data manager updates the state when we call `tpc_vote`, after calling `commit`:

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

## 6.2 The `tpc_begin` Method

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

```
>>> rm = DataManager()
>>> 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 preceding savepoint, the transaction must match:

```

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

```

## 6.3 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.

## 6.4 The `tpc_finish` Method

Complete two-phase commit

```

>>> rm = DataManager()
>>> rm.state
0
>>> rm.inc()

We start two-phase commit by calling ``tpc_begin``, ``commit``, and ``tpc_vote``:

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

We complete it by calling tpc_finish:

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

```

It is an error to 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 mismatch', '2', '3')
```

## 6.5 The `tpc_abort` Method

Abort a transaction during two-phase commit *after* `tpc_vote` has been called.

Here, we will ignore the fact that this is only called after `tpc_vote` and simulate that using `inc`.

```
>>> rm = DataManager()
>>> 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 mismatch', '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)
```

(continues on next page)



(continued from previous page)

```

>>> 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)
Traceback (most recent call last):
...
TypeError: ('Transaction mismatch', '2', '1')
>>> rm.tpc_abort(t1)

```

This should never fail.

## 6.6 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
(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):
...

```

(continues on next page)

(continued from previous page)

```
TypeError: ('Transaction mismatch', '2', '1')
>>> dm.abort(t1)
```

Of course, the transactions passed to abort must match. (Since it's called before `tpc_vote` is called, there might be no current transaction.)

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

## 6.7 The commit method

Called after `tpc_begin` to make changes persistent and prepare for voting.

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

We start two-phase commit by calling `tpc_begin`

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

We complete it by calling `commit`, `tpc_vote`, and `tpc_finish`:

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

It is an error to call `commit` without calling `tpc_begin` first:

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

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

```
>>> dm = DataManager()
>>> t3 = '3'
>>> dm.tpc_begin(t3)
>>> dm.commit(t2)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: ('Transaction mismatch', '2', '3')
```

## 6.8 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.

```
>>> rm = DataManager()
>>> 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 mismatch', '2', '1')
```

```
>>> 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
```

---

## Transaction integrations / Data Manager Implementations

---

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



---

## 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.

### 8.1 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>.

### 8.2 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.

## 8.3 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:



```

1 >>> import transaction
2 >>> session.add(User(id=1, name='John', fullname='John Smith', password='123'))
3 >>> 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.

## 8.4 Aborting transactions

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

```

1 >>> session = Session()
2 >>> john = session.query(User).all()[0]
3 >>> john.fullname
4 u'John Smith'
5 >>> john.fullname = 'John Q. Public'
6 >>> john.fullname
7 u'John Q. Public'
8 >>> 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:

```

1 >>> session = Session()
2 >>> john = session.query(User).all()[0]
3 >>> john.fullname
4 u'John Smith'

```

## 8.5 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:

```

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

```

(continues on next page)

(continued from previous page)

```
6 >>> Base.metadata.create_all(engine)
7 >>> class User(Base):
8 ...     __tablename__ = 'users'
9 ...     id = Column(Integer, primary_key=True)
10 ...     name = Column(String)
11 ...     fullname = Column(String)
12 ...     password = Column(String)
13 ...
14 >>> Base.metadata.create_all(engine)
15 >>> from sqlalchemy.orm import sessionmaker
16 >>> from zope.sqlalchemy import ZopeTransactionExtension
17 >>> Session = sessionmaker(bind=engine, extension=ZopeTransactionExtension())
```

We are now ready to create and use a savepoint:

```
1 >>> import transaction
2 >>> session = Session()
3 >>> session.add(User(id=1, name='John', fullname='John Smith', password='123'))
4 >>> 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 of course we have to hold on to it in case we do.

Now, we'll add a second user:

```
1 >>> session.add(User(id=2, name='John', fullname='John Watson', password='123'))
2 >>> [o.fullname for o in session.query(User).all()]
3 [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:

```
1 >>> sp.rollback()
2 >>> [o.fullname for o in session.query(User).all()]
3 [u'John Smith']
4 >>> 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.

---

## 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:

```
1 >>> from sqlalchemy import create_engine
2 >>> engine = create_engine('postgresql://postgres@127.0.0.1:5432')
3 >>> from sqlalchemy import Column, Integer, String
4 >>> from sqlalchemy.ext.declarative import declarative_base
5 >>> Base = declarative_base()
6 >>> Base.metadata.create_all(engine)
7 >>> class User(Base):
8 ...     __tablename__ = 'users'
9 ...     id = Column(Integer, primary_key=True)
10 ...     name = Column(String)
11 ...     fullname = Column(String)
12 ...     password = Column(String)
13 ...
14 >>> Base.metadata.create_all(engine)
15 >>> from sqlalchemy.orm import sessionmaker
16 >>> from zope.sqlalchemy import ZopeTransactionExtension
17 >>> Session = sessionmaker(bind=engine, extension=ZopeTransactionExtension())
```

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

```
1 >>> from ZODB import DB, FileStorage
2
3 >>> storage = FileStorage.FileStorage('test.fs')
4 >>> db = DB(storage)
5 >>> connection = db.open()
6 >>> 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:

```
1 >>> import transaction
2 >>> session.add(User(id=1, name='John', fullname='John Smith', password='123'))
3 >>> 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.

## 10.1 Interfaces

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

An object that manages a sequence of transactions.

Applications use transaction managers to establish transaction boundaries.

A transaction manager supports the “context manager” protocol: Its `__enter__` begins a new transaction; its `__exit__` commits the current transaction if no exception has occurred; otherwise, it aborts it.

**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`.

New in version 2.1.0.

**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 ()**

Return 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 *ISynchronizer* objects.

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

**attempts (number=3)**

Generate up to *number* (transactional) context managers.

This method is typically used as follows:

```
for attempt in transaction_manager.attempts():
    with attempt:
        *with block*
```

The `with attempt:` starts a new transaction for the execution of the `with block`. If the execution succeeds, the (then current) transaction is committed and the `for` loop terminates. If the execution raised an exception, then the transaction is aborted. If the exception was some kind of `retriable error` and the maximal number of attempts is not yet reached, then a next iteration of the `for` loop starts. In all other cases, the `for` loop terminates with the exception.

**run** (*func=None, tries=3*)

Call `func()` in its own transaction; retry in case of some kind of `retriable error`.

The call is tried up to `tries` times.

The call is performed in a new transaction. After the call, the (then current) transaction is committed (no exception) or aborted (exception).

`run` supports the alternative signature `run(tries=3)`. If `func` is not given or passed as `None`, then the call to `run` returns a function taking `func` as argument and then calling `run(func, tries)`.

**interface** `transaction.interfaces.ITransaction`

Object representing a running transaction.

**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 `DoomedTransaction` 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 *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.

#### Parameters

- **name** (*text*) – is the text (unicode) name of the extension property to set
- **value** – must be picklable and json serializable

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.

#### Parameters

- **args** (*sequence*) – Additional positional arguments to be passed to the hook. The default is to pass no positional arguments.
- **kws** (*dict*) – Keyword arguments to pass to the hook. The default is to pass no keyword arguments.

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 *ITransactionManager.registerSynch* instead.

**getBeforeCommitHooks** ()

Return iterable producing registered *addBeforeCommitHook* 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* and *kws* are interpreted as for `addBeforeCommitHook` (with the exception that there is always one positional argument, the commit status). As with `addBeforeCommitHook`, multiple hooks can be registered, savepoint creation doesn't call any hooks, and calling a hook consumes its registration.

**getAfterCommitHooks** ()

Return iterable producing the registered `addAfterCommitHook` hooks.

As with `getBeforeCommitHooks`, 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.

**addBeforeAbortHook** (*hook*, *args=()*, *kws=None*)

Register a hook to call before the transaction is aborted.

The specified hook function will be called after the transaction's abort method has been called, but before the abort process has been started.

*args* and *kws* are interpreted as for `addBeforeCommitHook`. As with `addBeforeCommitHook`, multiple hooks can be registered, savepoint creation doesn't call any hooks, and calling a hook consumes its registration.

Abort hooks are called only for a top-level abort. If the transaction is committed, abort hooks are not called. This is true even if the commit fails. In this case, however, the transaction is in the `COMMITFAILED` state and is virtually unusable; therefore, a top-level abort will typically follow.

**getBeforeAbortHooks** ()

Return iterable producing the registered `addBeforeAbortHook` hooks.

As with `getBeforeCommitHooks`, 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 abort.

**addAfterAbortHook** (*hook*, *args=()*, *kws=None*)

Register a hook to call after a transaction abort.

The specified hook function will be called after the transaction abort.

*args* and *kws* are interpreted as for `addBeforeCommitHook`. As with `addBeforeCommitHook`, multiple hooks can be registered, savepoint creation doesn't call any hooks, and calling a hook consumes its registration.

As with `addBeforeAbortHook`, these hooks are called only for a top-level abort. See that method for more.

**getAfterAbortHooks** ()

Return iterable producing the registered `addAfterAbortHook` hooks.

As with `getBeforeCommitHooks`, 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 abort.

**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 a 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.

Returns true if any joined *IRetryDataManager* considers the error transient *or* if the error is an instance of *TransientError*. 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.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.

These objects implement the two-phase commit protocol in order to allow multiple data managers to safely participate in a single transaction. The methods *tpc\_begin*, *commit*, *tpc\_vote*, and then either *tpc\_finish* or *tpc\_abort* are normally called in that order when committing a 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 *IDataManagers*.

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

Transactions use a global sort order to prevent deadlock when committing transactions involving multiple data managers. The data managers **must** define a *sortKey* method that provides a global ordering across all registered data 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 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

**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 explicitly.

## 10.1.1 Exceptions

**class** `transaction.interfaces.TransactionError`

Bases: `exceptions.Exception`

An error occurred due to normal transaction processing.

**class** `transaction.interfaces.TransactionFailedError`

Bases: `transaction.interfaces.TransactionError`

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 by invoking *ITransaction.abort*. (If the transaction manager is not operating in explicit mode, then *ITransactionManager.begin* can also be used to perform an implicit abort.)

**class** `transaction.interfaces.DoomedTransaction`

Bases: `transaction.interfaces.TransactionError`

A commit was attempted on a transaction that was doomed.

**class** `transaction.interfaces.TransientError`

Bases: `transaction.interfaces.TransactionError`

An error has occurred when performing a transaction.

It's possible that retrying the transaction will succeed.

**class** `transaction.interfaces.InvalidSavepointRollbackError`

Bases: `exceptions.Exception`

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.

**class** `transaction.interfaces.NoTransaction`

Bases: `transaction.interfaces.TransactionError`

No transaction has been defined

An application called an operation on a transaction manager that affects an existing transaction, but no transaction was begun. The transaction manager was in explicit mode, so a new transaction was not explicitly created.

New in version 2.1.0.

**class** `transaction.interfaces.AlreadyInTransaction`

Bases: `transaction.interfaces.TransactionError`

Attempt to create a new transaction without ending a preceding one

An application called `begin` on a transaction manager in explicit mode, without committing or aborting the previous transaction.

New in version 2.1.0.

## 10.2 API Objects

`transaction` module: Exported transaction functions.

`transaction.manager` = `<transaction._manager.ThreadTransactionManager object>`

The default transaction manager (a `ThreadTransactionManager`). All other functions in this module refer to this object.

`transaction.get` = `<bound method ThreadTransactionManager.get of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.get`

`transaction.begin` = `<bound method ThreadTransactionManager.begin of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.begin`

`transaction.commit` = `<bound method ThreadTransactionManager.commit of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.commit`

`transaction.abort` = `<bound method ThreadTransactionManager.abort of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.abort`

`transaction.doom` = `<bound method ThreadTransactionManager.doom of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.doom`

`transaction.isDoomed` = `<bound method ThreadTransactionManager.isDoomed of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.isDoomed`

`transaction.savepoint` = `<bound method ThreadTransactionManager.savepoint of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.savepoint`

`transaction.attempts` = `<bound method ThreadTransactionManager.attempts of <transaction._manager.ThreadTransactionManager object>>`

See `ITransactionManager.attempts`

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

Bases: `object`

Default implementation of `ITransaction`.

`isDoomed()`

See `ITransaction`.

`doom()`

See `ITransaction`.

`join(resource)`

See `ITransaction`.

`savepoint(optimistic=False)`

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*.

**getBeforeAbortHooks** ()  
See *ITransaction*.

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

**getAfterAbortHooks** ()  
See *ITransaction*.

**addAfterAbortHook** (*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.TransactionManager` (*explicit=False*)

Bases: `object`

Single-thread implementation of *ITransactionManager*.

**\_\_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.**ThreadTransactionManager**

Bases: `thread._local`

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.

**class** transaction.**Savepoint** (*transaction, optimistic, \*resources*)

Bases: `object`

Implementation of *ISavepoint*, a transaction savepoint.

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

**rollback ()**  
See *ISavepoint*.





Transaction objects manage resources for an individual activity. This document contains some notes that will help in understanding how transactions work, and how to use them to accomplish specific objectives.

### 11.1 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 *IDataManager.tpc\_begin* on each resource manager before calling *IDataManager.commit* on any of them.

1. *tpc\_begin(txn)*
2. *commit(txn)*
3. *tpc\_vote(txn)*
4. *tpc\_finish(txn)*

### 11.2 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 *ITransaction.addBeforeCommitHook*, passing it a callable and arguments. The callable will be called with its arguments at the start of the commit.

### 11.3 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 `ITransaction.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 successful or false iff aborted) preceding its arguments at the start of the commit.

## 11.4 Abort hooks

Commit hooks are not called for `ITransaction.abort`. For that, use `ITransaction.addBeforeAbortHook` or `ITransaction.addAfterAbortHook`.

## 11.5 Error handling

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

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

Once uncommitted objects are aborted, `IDataManager.tpc_abort` is called on each data manager.

## 11.6 Transaction Manager Lifecycle Notifications (Synchronization)

You can register synchronization objects (*synchronizers*) with the transaction manager. The synchronizer must implement `ISynchronizer.beforeCompletion` and `ISynchronizer.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 `ITransaction` as their only argument.

## 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.



## CHAPTER 12

---

### 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 13

---

## 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:

```
1 import transaction
2
3 try:
4     # some code that uses one or more backends
5     .
6     .
7     .
8     transaction.commit()
9 except SomeError:
10    transaction.abort()
```





---

## Things you need to know about the transaction machinery

---

### 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.

### 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 its own copy.<sup>1</sup>

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

### 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.

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.

---

<sup>1</sup> 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.

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.

### 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.

### Savepoints

A savepoint allows *supported data managers* to save work to their 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.

**t**

`transaction`, [57](#)

`transaction.interfaces`, [49](#)



---

## Symbols

`__enter__()` (*transaction.TransactionManager method*), 58  
`__exit__()` (*transaction.TransactionManager method*), 58

### A

`abort` (*in module transaction*), 57  
`abort()` (*transaction.interfaces.IDataManager method*), 54  
`abort()` (*transaction.interfaces.ITransaction method*), 51  
`abort()` (*transaction.interfaces.ITransactionManager method*), 50  
`abort()` (*transaction.Transaction method*), 58  
`abort()` (*transaction.TransactionManager method*), 59  
`addAfterAbortHook()` (*transaction.interfaces.ITransaction method*), 53  
`addAfterAbortHook()` (*transaction.Transaction method*), 58  
`addAfterCommitHook()` (*transaction.interfaces.ITransaction method*), 52  
`addAfterCommitHook()` (*transaction.Transaction method*), 58  
`addBeforeAbortHook()` (*transaction.interfaces.ITransaction method*), 53  
`addBeforeAbortHook()` (*transaction.Transaction method*), 58  
`addBeforeCommitHook()` (*transaction.interfaces.ITransaction method*), 52  
`addBeforeCommitHook()` (*transaction.Transaction method*), 58  
`afterCompletion()` (*transaction.interfaces.ISynchronizer method*), 56  
`AlreadyInTransaction` (*class in transaction.interfaces*), 57  
`attempts` (*in module transaction*), 57  
`attempts()` (*transaction.interfaces.ITransactionManager method*),

50

### B

`beforeCompletion()` (*transaction.interfaces.ISynchronizer method*), 56  
`begin` (*in module transaction*), 57  
`begin()` (*transaction.interfaces.ITransactionManager method*), 49  
`begin()` (*transaction.TransactionManager method*), 58

### C

`clearSynchs()` (*transaction.interfaces.ITransactionManager method*), 50  
`clearSynchs()` (*transaction.TransactionManager method*), 58  
`commit` (*in module transaction*), 57  
`commit()` (*transaction.interfaces.IDataManager method*), 54  
`commit()` (*transaction.interfaces.ITransaction method*), 51  
`commit()` (*transaction.interfaces.ITransactionManager method*), 49  
`commit()` (*transaction.Transaction method*), 57  
`commit()` (*transaction.TransactionManager method*), 59

### D

`data()` (*transaction.interfaces.ITransaction method*), 54  
`description` (*transaction.interfaces.ITransaction attribute*), 51  
`doom` (*in module transaction*), 57  
`doom()` (*transaction.interfaces.ITransaction method*), 51  
`doom()` (*transaction.interfaces.ITransactionManager method*), 50  
`doom()` (*transaction.Transaction method*), 57  
`doom()` (*transaction.TransactionManager method*), 59

DoomedTransaction (class in transaction.interfaces), 56

## E

explicit (transaction.interfaces.ITransactionManager attribute), 49

extension (transaction.interfaces.ITransaction attribute), 51

## G

get (in module transaction), 57

get () (transaction.interfaces.ITransactionManager method), 49

get () (transaction.TransactionManager method), 58

getAfterAbortHooks () (transaction.interfaces.ITransaction method), 53

getAfterAbortHooks () (transaction.Transaction method), 58

getAfterCommitHooks () (transaction.interfaces.ITransaction method), 53

getAfterCommitHooks () (transaction.Transaction method), 58

getBeforeAbortHooks () (transaction.interfaces.ITransaction method), 53

getBeforeAbortHooks () (transaction.Transaction method), 58

getBeforeCommitHooks () (transaction.interfaces.ITransaction method), 52

getBeforeCommitHooks () (transaction.Transaction method), 58

## I

IDataManager (interface in transaction.interfaces), 54

IDataManagerSavepoint (interface in transaction.interfaces), 55

InvalidSavepointRollbackError (class in transaction.interfaces), 56

IRetryDataManager (interface in transaction.interfaces), 55

ISavepoint (interface in transaction.interfaces), 55

ISavepointDataManager (interface in transaction.interfaces), 55

isDoomed (in module transaction), 57

isDoomed () (transaction.interfaces.ITransactionManager method), 50

isDoomed () (transaction.Transaction method), 57

isDoomed () (transaction.TransactionManager method), 59

isRetryableError () (transaction.interfaces.ITransaction method), 54

ISynchronizer (interface in transaction.interfaces), 56

ITransaction (interface in transaction.interfaces), 51

ITransactionManager (interface in transaction.interfaces), 49

## J

join () (transaction.interfaces.ITransaction method), 52

join () (transaction.Transaction method), 57

## M

manager (in module transaction), 57

## N

newTransaction () (transaction.interfaces.ISynchronizer method), 56

note () (transaction.interfaces.ITransaction method), 52

note () (transaction.Transaction method), 58

NoTransaction (class in transaction.interfaces), 56

## R

registeredSynchs () (transaction.interfaces.ITransactionManager method), 50

registeredSynchs () (transaction.TransactionManager method), 58

registerSynch () (transaction.interfaces.ITransactionManager method), 50

registerSynch () (transaction.TransactionManager method), 58

rollback () (transaction.interfaces.IDataManagerSavepoint method), 55

rollback () (transaction.interfaces.ISavepoint method), 55

rollback () (transaction.Savepoint method), 59

run () (transaction.interfaces.ITransactionManager method), 51

## S

Savepoint (class in transaction), 59

savepoint (in module transaction), 57

savepoint () (transaction.interfaces.ISavepointDataManager method), 55

savepoint () (transaction.interfaces.ITransaction method), 51

savepoint () (transaction.interfaces.ITransactionManager method), 50

savepoint () (transaction.Transaction method), 57

savepoint () (transaction.TransactionManager method), 59

set\_data() (*transaction.interfaces.ITransaction method*), 53  
 setExtendedInfo() (*transaction.interfaces.ITransaction method*), 52  
 setExtendedInfo() (*transaction.Transaction method*), 58  
 setUser() (*transaction.Transaction method*), 58  
 should\_retry() (*transaction.interfaces.IRetryDataManager method*), 55  
 sortKey() (*transaction.interfaces.IDataManager method*), 55

## T

ThreadTransactionManager (*class in transaction*), 59  
 tpc\_abort() (*transaction.interfaces.IDataManager method*), 55  
 tpc\_begin() (*transaction.interfaces.IDataManager method*), 54  
 tpc\_finish() (*transaction.interfaces.IDataManager method*), 54  
 tpc\_vote() (*transaction.interfaces.IDataManager method*), 54  
 Transaction (*class in transaction*), 57  
 transaction (*module*), 57  
 transaction.interfaces (*module*), 49  
 transaction\_manager (*transaction.interfaces.IDataManager attribute*), 54  
 TransactionError (*class in transaction.interfaces*), 56  
 TransactionFailedError (*class in transaction.interfaces*), 56  
 TransactionManager (*class in transaction*), 58  
 TransientError (*class in transaction.interfaces*), 56

## U

unregisterSynch() (*transaction.interfaces.ITransactionManager method*), 50  
 unregisterSynch() (*transaction.TransactionManager method*), 58  
 user (*transaction.interfaces.ITransaction attribute*), 51

## V

valid (*transaction.interfaces.ISavepoint attribute*), 55