ZODB Book Documentation

Release 1

Carlos de la Guardia and the Zope community

Sep 27, 2017

Contents

This book is a work in progress. These documents discuss the objectives:

CHAPTER 1

ZODB Book Outline

Part one: getting started

This part will have an emphasis on getting an application up and running while making simple use of the ZODB. A developer who just needs to add a simple persistent layer to his application might have enough with this.

Introduction to ZODB

This will be a very short chapter, just to get things going. What is the ZODB. Maybe some bits about the NoSQL craze, how the ZODB has been doing that for more than 10 years. Why is the ZODB a nice tool to keep in your Python developer's arsenal and when is it a good fit for your apps?

Your first ZODB application

Installation and running the first app. The objective of this chapter is to let the reader do something that works immediately. Just the basics to get an app running. Not a lot of details here.

Working with the ZODB

A bit more involved explanation of how the ZODB works and a more useful sample application. This chapter will cover usage of the ZODB in detail.

Transactions

The ZODB depends on the transaction package and understanding this package is very important to working effectively with it. This chapter introduces transactions, shows what happens when you commit or abort, describes what a conflict error is and explains why it's a good idea to avoid long running transactions.

Basic indexing and searching

The Catalog and indexes. I propose to use repoze.catalog here, which uses zope.index.

ZODB patterns

This chapter shows some common patterns for working with specific situations that come up when using the ZODB.

Maintenance

Packing, what it is, why it can take a long time, how garbage collection affects it. Automated packing. Backing up, automated back ups.

Scaling

The ZODB cache, ZEO and replication services.

Part two - advanced topics

This will be a more in-depth review of techniques and concepts for ZODB development.

A more in-depth look at the ZODB internals

A little more information about how the ZODB works. At least enough stuff to understand the later chapters about storages and debugging.

Advanced transaction management

How to create data managers for working with other storages in the same transaction, how to best approach the need for well behaved, long running transactions.

ZODB Storages

Details about the FS storage and discussion of RelStorage and maybe DirectoryStorage. Different packing strategies across various storages.

Popular third party packages

Some of the most important packages for the ZODB will be described here.

Other indexing and searching strategies

Other catalog implementations, third party indexes and using external indexing solutions, like Solr.

Advanced ZODB

Evolving schemas, creating custom indexes.

The debugging FAQ: frequent problems and suggested solutions

General debugging strategies and then a FAQ with common problems. For example, common traps like attempting to load an object state when the connection is closed.

Part three - ZODB API

The official public API will be documented here. This could serve as a really quick reference for developers. We might include APIs for some other modules, like transaction.

CHAPTER 2

Notes about intended audience

Due to the Python-centric nature of the ZODB, familiarity with Python will be assumed for this book. This means that we will have no sections on what is Python, Python's strenghts and installation.

At this point, I think it's also safe to assume that PyPi will be known to prospective ZODB developers, thus no easy_install explanation either.

The intended audience might be defined therefore as Python developers who are looking for an easy way to store their application's information.

This is an open license project. Anyone can download and use the source. Make a clone of the repository today, using our Github repository.

For more information, visit the ZODB Documentation Blog.

Contents:

CHAPTER 3

Introduction

Imagine: you are writing a Python application of some kind. You have defined a number of complex objects, each with properties of various types. Some of these properties are other objects that are also defined within part of your code. You now need to persist the state of these objects, so that this state can be retrieved in future runs of your program. The Zope Object Database ("ZODB") can help. The ZODB allows you to persist your Python objects easily.

Python's dynamic nature allows developers to quickly develop applications, avoiding the compile cycle and static typing declarations required by other languages. The ZODB offers a similar benefit: developers who use the ZODB can store their objects transparently without any cumbersome mapping of objets to relational database tables. They no longer need to worry about ways of decomposing complex objects in order to fit a relational or filesystem model: using ZODB, programmers can store Python objects in their native, assembled state.

This book will introduce the ZODB in detail and will cover its features step-by-step, using a hands-on approach with many code examples and practical tips.

Looking at the ZODB from 10,000 feet

ZODB takes a minimalist approach, compared to other systems which call themselves "databases". ZODB provides persistence and transaction support, but provides no security, indexing or search facilities. There are third party packages which provide these (and additional) services for the ZODB, but for now let's take a look at its core features.

Familiarity

For a Python developer, the ZODB offers a very familiar environment. Not only does it store native Python objects, but even its internal serialization mechanism is well known to Python programmers: it uses the *pickle* module included in the Python Standard Library. ZODB is written in Python itself, so in extreme cases developers can look under the hood without needing other tools than the ones they use regularly.

Simplicity

The ZODB is a hierarchical database. There is a root object, initialized when a database is created. The root object is used like a Python dictionary and it can contain other objects (which can be dictionary-like themselves). To store an object in the database, it's enough to assign it to a new key inside its container.

Production-readiness

The ZODB has been around for over ten years and has been put into many production environments.

Transparency

To make an instance of a class automatically persistent, it's enough for its class to inherit from a base "Persistent" class. The ZODB will then take care of saving objects which inherit from this class whenever they are changed. Non-persistent objects can also be saved easily to ZODB, but in some cases it becomes necessary to alert the ZODB when they change.

Transaction support

Transactions are a series of changes to the database that need to be carried out as a unit. That is, either all of the changes in a transaction take place, or none do. Generally, when you are through with a series of changes the transaction is *committed* and if anything goes wrong it is *aborted*.

If you have worked with relational databases, transactions should be familiar. Transactional systems need to make sure that the database never gets into an inconsistent state, which they do by supporting four properties, known by the acronym ACID:

- Atomicity. Either all the modifications grouped in a transaction will be written to the database or, if something makes this impossible, the whole transaction will be aborted. This insures that in the event of a write error or a hardware glitch the database will remain in the previous state and avoid inconsistencies.
- Consistency. For writing transactions, these means that no transaction will be allowed if it would leave the database in an inconsistent state. For reading transactions it means that a read operation will see the database in the consistent state it was at the beginning of the transaction, regardless of other transactions taking place at the time.
- Isolation. When changes are made to the database by two different programs, they will not be able to see each other's transactions until they commit their own.
- Durability. This simply means that the data will be safely stored once the transaction is committed. A software or hardware crash will not cause any information to be lost after that.

Save points

Since changes made during a single transaction are kept in memory until the transaction is committed, memory usage can skyrocket during a transaction where lots of objects are modified at the same time (say a for loop which changes a property on 100,000 objects). Save points allow developers to commit part of one transaction before it's finished so that changes are written to the database and the memory they occupied is released. This changes in the save point are not committed until the whole transaction is finished, so that if it is aborted, any save points will be rolled back as well.

Undo

The ZODB provides a very simple mechanism to roll back any committed transaction. This feature is possible because ZODB keeps track of the database state before and after every transaction. This makes it possible to undo the changes in a transaction, even if more transactions have been committed after it. Of course, if the objects involved in this transaction that we need to undo have changed in later transactions, it will not be possible to undo it because of consistency requirements.

History

Since every transaction is kept in the database, it's possible to view an object's state as it was in previous transactions and compare it with its current state. This allows a developer to quickly implement simple versioning functionality.

Blobs

Binary large objects, such as images or office documents, do not need all the versioning facilities that the ZODB offers. In fact, if they were handled as regular object properties, blobs would make the size of a database increase greatly and generally slow things down. That's why the ZODB uses a special storage for blobs, which makes it feasible to easily handle large files up to a few hundred megabytes without performance problems.

In-memory caching

Every time an object is read from the database, it's kept in an in-memory LRU cache. Subsequent accesses to this object consume less resources and time. The ZODB manages the cache transparently and automatically pulls out objects which have not been accessed for a long time. The size of the cache can be configured, so that machines with more memory can take better advantage of the feature.

Packing

ZODB keeps all versions of the objects stored in it. This means that the database grows with every object modification and it can reach a large size, which may slow it down and consume more space than is necessary. The ZODB allows us to remove old revisions of stored objects via a procedure known as *packing*. The packing routine is flexible enough to allow only objects older than a specified number of days to be removed, keeping the newer revisions around.

Pluggable storages

By default, the ZODB stores the database in a single file. The program which manages this is called a file storage. However, the ZODB is built in such a way that other storages can be plugged in without needing to modify its source code. This can be used to store ZODB data in other media or formats, as we'll see later in more detail.

Scalability

Zope Enterprise Objects (ZEO) is a network storage for the ZODB. Using ZEO, any number of ZODB clients can connect to the same ZODB. ZEO can be used to provide scalability because the load can be distributed between several ZEO clients instead of only one.

ZODB and relational databases

By far, the most popular mechanism for storing program data is a relational database. The relational model uses tables, which contain data that conforms to a predefined schema. Each table column represents an attribute of the schema and each row is a set of values for those columns. The power of this model comes from the ability to relate tables by one or more common attributes, so that data can be queried and assembled in multiple ways.

Relational databases are widely popular, in part because their programming language independece makes them relatively easy to use in a variety of work environments. They usually require specific drivers for any given programming language, but that's not really a problem in the case of Python, as it has bindings to all the major relational databases (and some minor ones).

Of course, the vast majority of relational databases do not natively store Python objects, so it's necessary for the application itself to read the data from the tables and "assemble" the columns from each row into the required objects. The application is also responsible for breaking apart the objects and fitting their attributes into the table structure when a change is detected.

These assembly and disassembly is known as object/relational mapping and can use a significant portion of an application's logic. Fortunately, there are excellent third party Python packages, called ORMs (*object-relational mappers*), that take care of the interaction with the relational database for the application developer.

Using an ORM allows the developer to forget about the underlying database and focus on the Python objects, but the objects themselves most retain the tabular structure. Also, in many cases it's necessary to have a very good understanding of the specific database used and relational databases in general to be able to decide how best to structure the objects.

Working with the ZODB does not require any of this mapping activity, since the objects are stored in their native form, which simplifies the interaction between the developer and the data. Without the need for a tabular structure, data can better reflect the organization of information in the problem domain.

Instead of managing relations using different tables with common primary keys, the ZODB lets developers use normal Python object references. An object can be a "property" of a separate object without the need for table joins and multiple objects can reference this property without each actually having to store a copy of the object.

Because it works directly with Python objects, the ZODB doesn't require a pre-defined structure of columns and data types for the objects it stores, which means that object attributes can easily change both in quantity and type. This can often be a lot harder when using a relational database for storage.

One other advantage of using the ZODB over a relational database comes when the problem domain requires a filesystem-like structure. Modeling this kind of containment relationships does not come naturally for the relational model, but is quite easy with the hierarchichal nature of the ZODB. Content management systems are one example of an application domain that is very well suited for ZODB use.

Is the ZODB a NoSQL database?

In recent years, the term NoSQL has been consitently used to refer to a "new" breed of database systems which basically do not use the relational paradigm. Here is one semi-official definition of NoSQL, taken from http://nosql-database.org/:

"Next Generation Databases mostly addressing some of the points: being non- relational, distributed, open-source and horizontallly scalable. The original intention has been modern web-scale databases. The movement began early 2009 and is growing rapidly. Often more characteristics apply as: schema-free, easy replication support, simple API, eventually consistent / BASE (Basically Available, Soft state, Eventual consistency, or not ACID), and more."

The ZODB has been around for more than a decade and thus clearly predates this concept (as do most of the NoSQL databases in existence), but in the general sense it can be classified as a NoSQL database, because it shares the main characteristic of being non-relational.

The ZODB is also open source, horizontally scalable and schema-free, like many of its NoSQL counterparts. It is not distributed and does not offer easy replication, at least not for free.

ZODB != Zope

Zope is a web application server written in Python that has also been around for more than 10 years. Unlike most web frameworks, Zope encourages the use of an object database for persistence, rather than the usual relational database. The database used by Zope is, of course, the ZODB. The ZODB has been a vital part of Zope since Zope's creation, as you may have already guessed by its name.

In part for its strong association with Zope and probably also in part due to the low popularity of object databases in general, the ZODB is used very little outside of the Zope world. Developers without exposure to Zope tend to assume that you have to use one to get the other or are afraid that they would have to pull dozens of Zope dependencies if they chose to use the ZODB. Some might even believe that they have to write code in the 'Zope way' if they want to use it.

Part of the motivation for writing this book is to clearly show the wider Python world that the ZODB is a totally independent Python package that can be a much better fit than relational databases for data persistence in many Python projects. The ZODB is sufficiently transparent in use that you only need to follow a few very simple rules to get your application to store your objects. Everything else is "just Python".

CHAPTER 4

Our First ZODB Application

Now that we know a little about what the ZODB offers, let's see how to install it and use it. This chapter shows where and how to get the ZODB running with your Python installation and explains some basic concepts about the way that applications interact with the ZODB. Along the way, we'll create a simple ZODB application that stores a line drawing in the database.

Installing the ZODB from PyPI

The Python Package Index (PyPI) is a repository of software for Python, where thousands of packages are available for download.

What makes PyPI much more powerful is a Python script called easy_install —a script that allows Python developers to install any package indexed on the PyPI over the network, keeping track of dependencies and versions. Packages that can be easy installed are packaged either as compressed files or in a special format, using the .egg extension, and are known as Python eggs.

The easy_install module is a part of a package known as setuptools, so we need need to install that in order to obtain it. There's an installer for Windows and a .egg file for Unix/Linux/Mac available at the setuptools PyPI page on http://pypi.python.org/pypi/setuptools.

To install setuptools on Windows, just run the installer. For Unix/Linux/Mac systems, run the .egg file as a shell script, like in this example (yourxi version may vary):

```
$ sh setuptools-0.6c11-py2.4.egg
```

Many Linux distributions include a package for setuptools. In Ubuntu or Debian, for example, you can use apt-get to install it:

```
$ sudo apt-get install python-setuptools
```

However, we recommend that you install the newest version manually, even if there's a package available for your system, as this way you are assured of getting the latest version.

After this, the easy_install script will be available on the system Python's path, and the ZODB can be installed on your system using:

```
$ sudo easy_install ZODB3
```

Note that the package name is 'ZODB3', not just ZODB. The package and its few dependencies will be downloaded and installed. Before going any further, make sure that you can import the package using the Python interpreter:

```
$ python
>>> import ZODB
```

The line drawing application

We are ready to try out the ZODB for the first time. We'll do that by writing a small application and then adapting it to store some data using the ZODB.

One cool thing about Python is the huge number of packages available in the standard library. Let's use one of those to create a simple application to draw lines with the mouse on a canvas. All that you need is to have a full Python installation, including the Tkinter graphical toolkit. Windows and Mac OSX users should have no problem, but some linux distributions, like Ubuntu, require the python-tk package to be installed separately.

Here's the code for our application:

```
1  from turtle import *
2
3  def switchupdown(x=0, y=0):
4      pen()['pendown'] and not up() or down()
5
6  def clear():
7      clearscreen()
8      init()
9
10 def quit():
11     bye()
12
13 def init():
14     onscreenclick(goto,1)
15     onscreenclick(switchupdown,3)
16     onkey(quit, 'q')
17     onkey(clear, 'c')
18     listen()
19
20 if __name__ == "__main__":
21     init()
22     mainloop()
```

The turtle module provides graphics commands to move a turtle around a Tk canvas and draw dots along the way if the turtle's pen is "down". The module includes methods to listen to events like clicks and key presses, which we use here to create a small interactive demo.

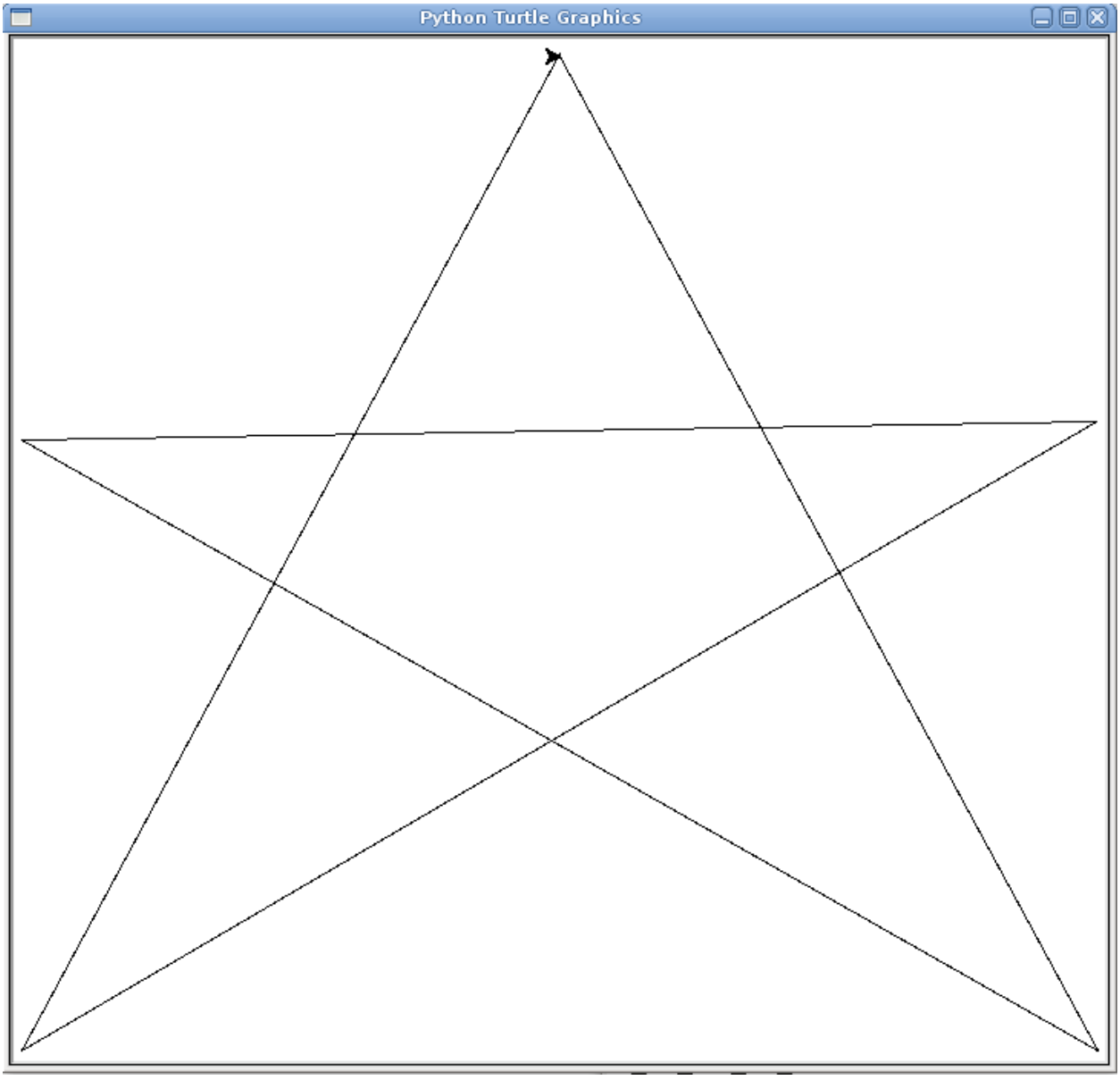
The code is fairly simple. When the program is run from the command line, the initialization routine is called on line 21 and then control is transferred to the Tk main loop to capture events and show the canvas window. The init method on lines 13-18 binds the events for mouse buttons and key presses to a couple of functions in our code and starts the listening process.

When the left mouse button is pressed, the goto function that is part of the turtle module is called. Tk events send mouse coordinates, which is precisely what goto needs, so the function can be bound directly to the event. The goto

function moves the turtle to the specified coordinates, drawing a line as it goes if the pen is down, as it is at the start of the program.

The right mouse button click event is bound to the switchupdown function on line 3, which toggles the up/down status of the pen. In addition, the 'c' key causes the clear function to be called to clear the screen and reinitialize the event bindings. Finally, the 'q' key calls the quit function, which just exits the program using the bye function from the turtle module.

The following image shows the program in action:



Saving drawings on the ZODB

Our overly simple drawing program works, though it would be nice if we could save our drawings to disk. Neither Tk nor turtle offer facilities for storing a drawing on disk and keep working on it later, which is what we'd like to do, so we decide to use the ZODB.

Turtle does not have a representation of the drawing that we can save, but it does have an undo buffer that stores every drawing command. Undo information is not kept inside a basic Python data structure. Instead, the turtle module has a Tbuffer class that allows pushing and popping drawing commands from it.

With most databases, storing the information from this buffer would require decomposing the Tbuffer instance into a compatible database representation, like table definitions. When reading the data back into the application, it would be necessary to create an instance of the Tbuffer class and manually set all its attributes to the stored values.

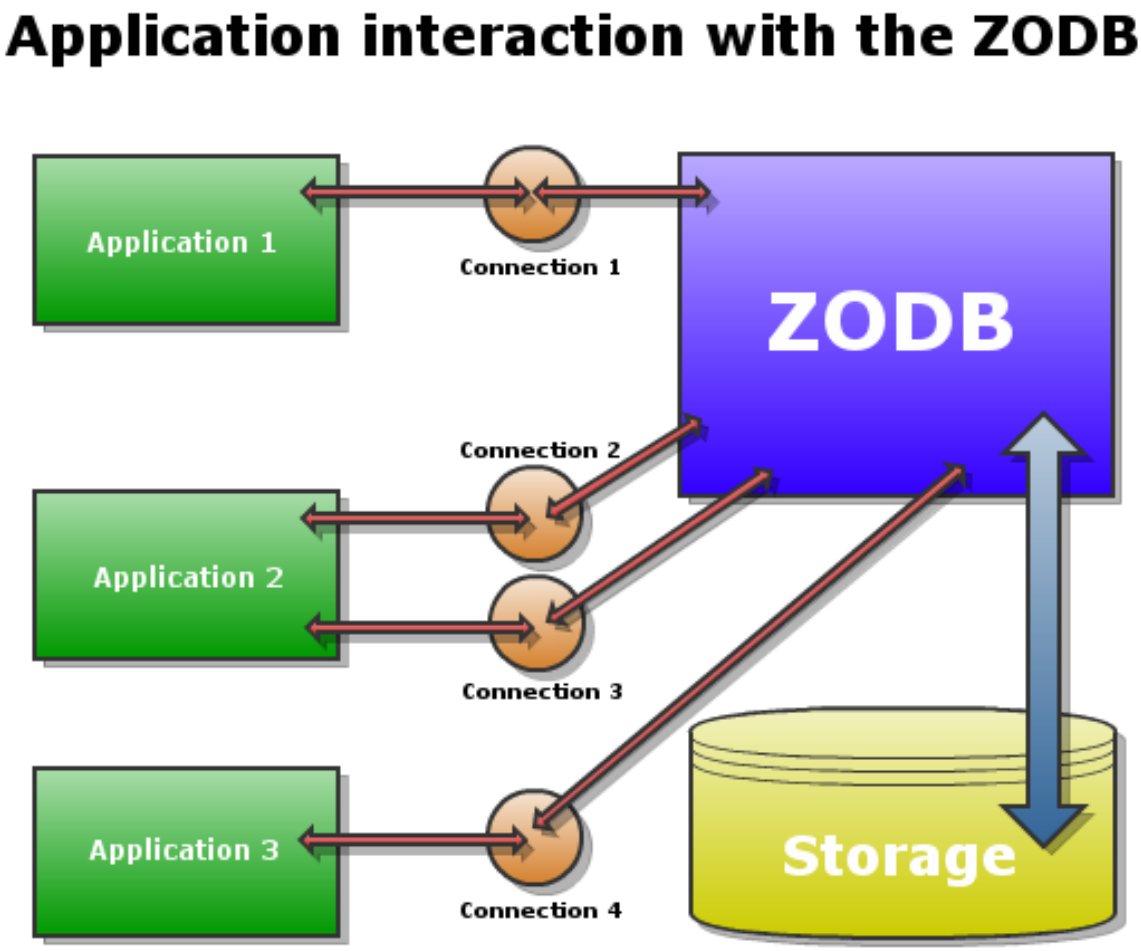
We mentioned in the introduction that the ZODB is mostly transparent and works with live Python objects. We also said that it's simple to use. Time to put those claims to the test.

Setting up a connection

The ZODB architecture defines a few concepts with which we need to be familiar before using it:

- A *storage* takes care of the low level details for handling the storage and retrieval of objects from some kind of store. By far the most used storage available for the ZODB is the FileStorage, which stores objects on a single file in the computer's file system. There are other kinds of storages, like RelStorage and DirectoryStorage, which store objects in a relational database and on multiple directories on a file system, respectively.
- A *database* is basically a process that handles a pool of connections. It is implemented by the DB class in the ZODB package.
- A *connection* provides access to the objects inside the database and takes care of caching them for faster access. A multi-threaded application can open one connection per thread and each connection will have its own cache and will be able to modify objects independently of the other threads.
- The *root* object of the database can be accessed through a connection and acts like a Python dictionary. Simple applications may not need anything more than this object to store their data. More complex applications can store complete hierarchies of containers and the ZODB package includes data structures specifically tailored to this task.

Thus, to connect to the ZODB a storage has to be opened and passed to a DB instance, which can handle one or more connections from the application. The following diagram illustrates this interactions:



For our drawing application, we'll use the default FileStorage, which takes a path to the database file as an initialization parameter. Let's add the following lines to the start of our turtle drawing program:

```
1 import transaction
2 from ZODB import DB, FileStorage
3
4 storage = FileStorage.FileStorage('drawing.fs')
5 db = DB(storage)
6 connection = db.open()
7 drawing = connection.root()
```

First we make a couple of imports. DB and FileStorage must be self-explanatory after the concepts that we have just discussed. The transaction module will be explained later.

On line 4 we create a FileStorage with the relative path 'drawing.fs'. This means that, if it does not exist, a file named drawing.fs will be created on the same directory as the program. If the path exists the database will append data to it. The filename used for the database can have any name that we choose. Usually the extension .fs is used, but it is by no means required.

Line 5 initializes the database using the storage that we just opened. Line 6 then opens a connection to this database. Finally, the root method of the connection instance is used to obtain the root object of the database. We are ready to start storing objects in there.

Storing the drawing on the ZODB

Earlier on, we had decided to store the undo buffer object on the database. Remember, the root object acts like a dictionary. To keep thing simple for now, let's just put the complete contents of the buffer under the key 'turtle_buffer' on the root. We'll do this only when the user decides to quit the application. The quit function will now look like this:

```
def quit():
    drawing['turtle_buffer'] = getturtle().undobuffer
    transaction.commit()
    bye()
```

Just before quitting, the program gets the undobuffer from the current turtle object and assigns it to the aforementioned turtle_buffer key of the drawing object, which represents the database root.

At this point the value of the turtle_buffer key will be the whole undobuffer, but the change will not be made permanent on the database until we commit the transaction. Between the time that we change something on the database and we commit the transaction, all the code in this thread will see the new value while other threads continue to work with the old.

In this case we commit the change immediately and exit the program. The full sequence of steps used to create the drawing that was on the screen will have been saved inside the ZODB.

The next step is to check if the turtle_buffer key exists in the root object at program start time, and if it does we can use its value to redraw the saved drawing. A couple of lines in the startup code will make this check:

```
if __name__ == "__main__":
    if 'turtle_buffer' in drawing:
        redraw(drawing['turtle_buffer'])
    init()
    mainloop()
```

All we need to do now is to repeat the undo steps in reverse order to get the saved drawing on the canvas. The redraw function takes advantage of some knowledge of the internal structure of the buffer to do this, but is otherwise straightforward. The final program with this function added looks like this:

```
from turtle import *

import transaction
from ZODB import DB, FileStorage

storage = FileStorage.FileStorage('drawing.fs')
db = DB(storage)
connection = db.open()
drawing = connection.root()

def switchupdown(x=0, y=0):
    pen()['pendown'] and not up() or down()

def redraw(turtle_buffer):
    ops = [turtle_buffer.pop() for i in range(turtle_buffer.nr_of_items())]
    ops.reverse()
    for op in ops:
        if op[0] == 'go':
            up()
            goto(op[1])
            if op[3][0]:
                down()
            goto(op[2])

```

```
def clear():
    clearscreen()
    init()

def quit():
    drawing['turtle_buffer'] = getturtle().undobuffer
    transaction.commit()
    bye()

def init():
    onscreenclick(goto,1)
    onscreenclick(switchupdown,3)
    onkey(quit, 'q')
    onkey(clear, 'c')
    listen()

if __name__ == "__main__":
    if 'turtle_buffer' in drawing:
        redraw(drawing['turtle_buffer'])
    init()
    mainloop()
```

That's it. Other than the transaction commit call, the code for storing the whole data structure for the buffer inside the ZODB is this:

```
drawing['turtle_buffer'] = getturtle().undobuffer
```

That's what we meant when we said that storing Python objects inside the ZODB is nearly transparent.

Why not just use pickle?

Some readers may be tempted to say that everything that we did above could just as easily have been done using the pickle module from the standard library.

They would be right, of course, since in the first place this was a very simple application and in the second place the ZODB uses pickle itself. However, this view misses the larger picture of the extra features that the ZODB offers beyond the simple serialization of objects and the huge difference this makes when working with larger and more complex applications.

Later sections of this book will elaborate a lot more on this topic, but for now let's finish the chapter by showing off one of these features, enabled by the transactional nature of the ZODB: the ability to undo transactions.

Undoing a transaction

Each time the user quits the drawing application, a transaction is committed that saves the current drawing inside the ZODB. The FileStorage storage that we use for this application supports history and undoing of transactions, so we could get back to any previous drawing if we needed to.

The current application has no support for this feature, but it's very easy to use the Python interactive interpreter to talk directly to the database and perform the undofrom there.

First, be sure to create at least a couple of drawings and remember the differences between the last two. For example, suppose that you drew a square and then quit the application, cleared the screen and drew a star. If you quit and restart the application now, you'll see the star. Quit again, then fire the Python interpreter:

```
$ python
>>> from ZODB import DB, FileStorage
>>> storage = FileStorage.FileStorage('drawing.fs')
>>> db = DB(storage)
```

We did the same imports that we have in our code and got a handle to the database. That's all we need to use undo. The undoInfo method of the database gives us all the transactions that we have made:

```
>>> db.undoInfo()
[{'description': '', 'size': 3929, 'user_name': '', 'id': 'A4jb//pPmRE=',
  'time': 1284435838.6666241},
 {'description': '', 'size': 3185, 'user_name': '', 'id': 'A4jb0VC8hMw=',
  'time': 1284433038.9225941},
 {'description': '', 'size': 4211, 'user_name': '', 'id': 'A4jLhw8D0qo=',
  'time': 1284182823.519125},
 {'description': '', 'size': 3376, 'user_name': '', 'id': 'A4jLhk2BBVU=',
  'time': 1284182778.1649971}]
```

This method gives us some interesting information for each transaction, like transaction size and time. In this case we are interested in the transaction id, since this is what the undo method requires to do its job:

```
>>> last_transaction = db.undoInfo()[0]
>>> db.undo(last_transaction['id'])
>>> import transaction
>>> transaction.commit()
```

The undo method of the database takes a transaction id and undoes the specified transaction. Notice that the undo operation is in itself a transaction and as such must be commited and can be undone as well. In this example, we undid the very last transaction (undoInfo shows transactions in inverse order).

If you close the interpreter session now and start the turtle application once more, the star drawing will be gone and the square will be shown on the canvas again.

Summary

In this chapter we covered how to install the ZODB package and created a very simple drawing application which stores the drawings on the ZODB. Of course, we have just scratched the surface of what the ZODB can do. Hopefully, you see the potential by now. In the next chapter we'll create a more realistic application and explain in detail how to work with the ZODB.

CHAPTER 5

Working with the ZODB

We got our feet wet in the last chapter with a simple toy program that showed how to save Python objects in the ZODB. In this chapter, we will learn the most important details about working with the ZODB by developing a more realistic application.

TimeTrax: A command line time tracking app

To get to know the ZODB better, we will develop an application to track the total hours used for a number of tasks, themselves organized in projects. This will be a command line application which allows the user to quickly give a project name, task and time to save in the database. It will also feature simple reporting for showing the total time worked per project and per task. TimeTrax will be a personal application, which means every user will need his own database.

Designing the application

Let's use a pretty simple design with a class each for projects, tasks and time bookings. A project can contain one or more tasks, a task can contain one or more time bookings. A booking is simply the time worked expressed in hours and a text description. That is all we need for now. The code for that could look like this:

```
class Project(object):
    def __init__(self, name, title):
        self.name = name
        self.title = title
        self.tasks = {}

    def addTask(self, name, description):
        task = Task(name, description)
        self.tasks[name] = task

class Task(object):
    def __init__(self, name,  description):
        self.name = name
```

```
14          self.description = description
15          self.bookings = []
16
17      def bookTime(self, time, description=''):
18          booking = Booking(time, description)
19          self.bookings.append(booking)
20
21  class Booking(object):
22      def __init__(self, time, description):
23          self.time = time
24          self.description = description
```

Project tasks will be stored in a dictionary because we need to be able to refer to them by name. Time bookings will be stored in a list, since they are just a collection of entries and we don't want order to matter just yet. We can add a date and time later if we decide that it's needed.

This is really all the code we need for the heart of our little app, but remember that we want to make it usable from the command line. By far the easiest way to do this is to use the cmd module from the Python standard library.

Integrating the cmd module

The command module provides a generic class to build line-oriented command interpreters. You might want to read the documentation for the module on the Python docs web site, but it's not essential to understand the ZODB part, which is what we are interested in.

We need to do two things to turn our app into a simple command shell:

1. Add a class that inherits from the cmd.Cmd class.
2. For every command that we want to define, add a method named do_<command>, which will be called when the user types that command in the shell.

As you'll see if you browse the complete code that we'll show at the end of the chapter, there are a couple of embellishments that can be added to a cmd.Cmd subclass. However, this is really all that's needed to make it work.

Let's create a class named TimeTrax that will be the main class for the application. In addition to being a subclass of cmd.Cmd, it will hold a number of projects in a dictionary. It will also allow callers to set the introductory text and command prompt for the shell.

```
1  import sys
2  import cmd
3  import shlex
4
5  class TimeTrax(cmd.Cmd, object):
6      def __init__(self, intro="TimeTrax time tracking helper",
7                   prompt="timetrax: "):
8          super(TimeTrax, self).__init__()
9          self.intro = intro
10         self.prompt = prompt
11         self.projects = {}
```

To use the cmd.Cmd class it's necessary to import the cmd module at the top. While we are at it, we'll add a couple of modules that will be used later. Notice that we inherit from cmd.Cmd and object. That is because Cmd is an old style class and we can't use the super call there in line 4 unless we inherit from object. Otherwise, the code should be self-explanatory.

Next, we'll add a few methods to deal with our time tracking classes:

```
    def addProject(self, name, title):
        project = Project(name, title)
        self.projects[name] = project

    def dropProject(self, name):
        del self.projects[name]

    def addTask(self, project, name, description):
        self.projects[project].addTask(name, description)

    def bookTime(self, project, task, time, description):
        self.projects[project].tasks[task].bookTime(time, description)
```

Again, the code should be self-explanatory, We are just working with instances of the classes that we defined previously and adding them to the TimeTrax projects dictionary.

We are now ready to add the actual shell commands, which will call the methods that we just defined:

```
def do_create(self, line):
    name, title = shlex.split(line)
    self.addProject(name, title)
    print "created project %s" % name

def do_drop(self, line):
    if line in self.projects.keys():
        self.dropProject(line)
        print "dropped project %s" % line
    else:
        print "'%s' is not a recognized project"

def do_add(self, line):
    project, task, description = shlex.split(line)
    self.addTask(project, task, description)
    print "Added task %s to project %s" % (task, project)

def do_book(self, line):
    args = shlex.split(line)
    if len(args) == 3:
        project, task, time= shlex.split(line)
        description = ''
    else:
        project, task, time, description = shlex.split(line)
    time = int(time)
    self.bookTime(project, task, time, description)
    print "booked %s hours for task %s in project %s" % (time,
                                                         task,
                                                         project)
```

The only missing part now is the reporting command. We want a single list command that takes care of listing projects, tasks and bookings, so we'll add three methods for doing the reporting, but only once command that will decide which method to call depending on the number of arguments received:

```
def do_list(self, line):
    if not line:
        return self.list_projects()
    args = shlex.split(line)
    if len(args) == 1:
        return self.list_tasks(line)
```

```
    return self.list_bookings(args[0], args[1])

def list_projects(self):
    print "Project             Title"
    print "=======             ====="
    for name, project in self.projects.items():
        print "%-20s%s" % (name, project.title)

def list_tasks(self, project):
    print "Project: %s" % project
    print
    print "Task                Time    Description"
    print "====                ====    ==========="
    for name, task in self.projects[project].tasks.items():
        total_time = sum([booking.time for booking in task.bookings])
        print "%-20s%-8s%s" % (name, total_time, task.description)

def list_bookings(self, project, task):
    task = self.projects[project].tasks[task]
    total_time = 0
    print "Project: %s" % project
    print "task: %s" % task.description
    print
    print "Time    Description"
    print "====    ==========="
    for booking in task.bookings:
        print "%-8s%s" % (booking.time, booking.description)
        total_time = total_time + booking.time
    print "----    -----------"
    print "%-8sTotal" % total_time
```

We are almost done. The last thing that we will do is that we want the shell to be run if there are no arguments passed to the TimeTrax program, but run a command directly if it's invoked in the command line. This code will take care of that:

```
if __name__ == '__main__':
    if len(sys.argv) > 1:
        args = [(' ' in arg and '"%s"' % arg) or arg for arg in sys.argv[1:]]
        TimeTrax().onecmd(' '.join(args))
    else:
        TimeTrax().cmdloop()
```

What we are doing here is that if any arguments are passed, we put quotes around any that include spaces and pass them off to the onecmd method of the cmd.Cmd superclass. This method, as it's name implies, just calls one command and returns control to the shell. When there are no arguments passed in, we simply hand control to the cmdloop method, which starts the TimeTrax interpreter up.

Running the TimeTrax application

We are now ready to test our program:

```
$ python TimeTrax.py
TimeTrax time tracking helper
timetrax: help

Documented commands (type help <topic>):
```

```
=======================================
EOF  add  book  create  drop  help  list
```

As explained above, if we call the program with no arguments, we'll get inside the shell. We are shown the introductory text and the timetrax prompt. The help command is useful for showing available commands. We can now create a project and add tasks to it:

```
timetrax: create timetraxdemo "TimeTrax demo project"

created project timetraxdemo

timetrax: add timetraxdemo code "Write TimeTrax code"

Added task code to project timetraxdemo

timetrax: add timetraxdemo docs "Write TimeTrax documentation"

Added task docs to project timetraxdemo
```

We can list projects and tasks:

```
timetrax: list

Project            Title
=======            =====
timetraxdemo        TimeTrax demo project

timetrax: list timetraxdemo

Project: timetraxdemo

Task                Time    Description
====                ====    ===========
docs                0       Write TimeTrax documentation
code                0       Write TimeTrax code
```

Of course, at this point the total time worked on both tasks is zero, since we haven't booked any hours. We'll do that now:

```
timetrax: book timetraxdemo code 1 "Wrote skeleton version of code"

booked 1 hours for task code in project timetraxdemo

timetrax: book timetraxdemo code 2 "Added cmd module and code"

booked 2 hours for task code in project timetraxdemo

timetrax: book timetraxdemo docs 2 "Wrote first draft of docs"

booked 2 hours for task docs in project timetraxdemo
```

We can now see the total time spent on each task or the details of all the time bookings for a task, using the list command:

```
timetrax: list timetraxdemo

Project: timetraxdemo
```

```
Task                Time    Description
====                ====    ===========
docs                2       Write TimeTrax documentation
code                3       Write TimeTrax code

timetrax: list timetraxdemo code

Project: timetraxdemo
task: Write TimeTrax code

Time    Description
====    ===========
1       Wrote skeleton version of code
2       Added cmd module and code
----    -----------
3       Total
```

The application works perfectly, but if we close the session all of our project information will be lost. Time to add some way to persist our data using the ZODB.

Storing the time tracking data in the ZODB

In the previous chapter we showed how easy it is to store a complex Python data structure inside the ZODB. Just assign the structure to a key in the ZODB root object and it will automatically be stored. This worked fine for such a small program, but imagine how hard it would be to continually write code to modify a specific attribute of a deeply nested list or dictionary.

Besides, no Python developer would construct his code in such a way. In the application we just wrote, we defined classes that represented the projects and tasks that we wanted to model. The most natural thing would be for these objects to save themselves automatically after every change. Luckily, the ZODB lets Python developers do just the natural thing most of the time.

In fact, there are only two basic things that we have to do to turn our TimeTrax program into a ZODB backed application:

1. Inherit from the persistent.Persistent class.
2. Commit a transaction after making a set of changes.

Of course, we would also need to open a connection to some ZODB database first, but hopefully you agree that these requirements are pretty transparent. Let's apply them to our TimeTrax code.

First, we need some imports at the top:

```
1 import persistent
2 import transaction
3 import ZODB
4 import ZODB.FileStorage
```

The first module, persistent, is the one that handles automatic notification of changes to the ZODB. The transaction package provides a general purpose transaction management system with two-phase commit which can be useful even without using the ZODB, as we'll learn in the next chapter. The other imports are the ZODB package and its FileStorage module, for storing the data in the file system.

The database connection will be open when we instantiate our class. We'll connect to the ZODB database and substitute the projects dictionary with a reference to the root object of the database:

```
class TimeTrax(cmd.Cmd, object):
    def __init__(self, intro="TimeTrax time tracking helper",
                 prompt="timetrax: ", db_path="projects.fs"):
        super(TimeTrax, self).__init__()
        self.intro = intro
        self.prompt = prompt
        self.db = ZODB.DB(ZODB.FileStorage.FileStorage(db_path))
        self.projects = self.db.open().root()
```

Notice that we also added a db_path parameter to the __init__ method, to allow the caller to use a different file for storing the database. By default, we will use the 'projects.fs' file in the working directory. Remember that the file will be created if it doesn't exist, so there will be no error message if the file is not where you think.

Next, we'll turn our Project, Task and Booking classes into persistent classes:

```
class Project(persistent.Persistent):
    def __init__(self, name, title):
        self.name = name
        self.title = title
        self.tasks = {}

    def addTask(self, name, description):
        task = Task(name, description)
        self.tasks[name] = task
        self._p_changed = True

class Task(persistent.Persistent):
    def __init__(self, name,  description):
        self.name = name
        self.description = description
        self.bookings = []

    def bookTime(self, time, description=''):
        booking = Booking(time, description)
        self.bookings.append(booking)
        self._p_changed = True

class Booking(persistent.Persistent):
    def __init__(self, time, description):
        self.time = time
        self.description = description
```

Mostly we inherit from persistent.Persistent instead of from object. We'll explain the odd self._p_changed assigment in lines 10 and 21 in a few moments.

Persistent object attributes can have any value that can be stored using the pickle protocol. This means that objects such as files or sockets can't be stored in the ZODB, because they are not pickleable.

All that's left is to commit a transaction at the end of every method that makes modifications to a project or its data:

```
def addProject(self, name, title):
    project = Project(name, title)
    self.projects[name] = project
    transaction.commit()

def dropProject(self, name):
    del self.projects[name]
    transaction.commit()

```

```
10 def addTask(self, project, name, description):
11     self.projects[project].addTask(name, description)
12     transaction.commit()
13
14 def bookTime(self, project, task, time, description):
15     self.projects[project].tasks[task].bookTime(time, description)
16     transaction.commit()
```

Other than the transaction.commit() calls, the methods are unchanged from the non-ZODB version of the code.

We are done. All in all, we added or modified 9 lines of code out of almost 200, plus the imports. Other than the class inheritance, we didn't have to change the program structure or have any special constraints or naming conventions applied to our model classes.

Try the program a few times and confirm that it now saves project data between sessions, then come back to read the fine print.

Handling changes to mutable objects

Now that we know the basic pattern for working with the ZODB, we should come back to the self._p_changed assignment that we said we'd explain later. Take a look at the addProject method:

```
1 def addProject(self, name, title):
2     project = Project(name, title)
3     self.projects[name] = project
4     transaction.commit()
```

In this code, we create a project in line 2 and then store it in the root object of the ZODB in the following line. In line 4 we commit the transaction and the code is automatically stored in the database.

Because the root object is persistent, every one of its attributes is persisted automatically when the object changes. This includes non-persistent objects like lists, tuples or dictionaries. Most changes to a persistent object's attributes are automatically detected by the persistence machinery, but there is an exception: when a mutable attribute, such as a dictionary or a list is modified in place, the ZODB can't know about the change.

Let's look at it from the ZODB's point of view: to add an element to a list or assign a new key to a dictionary, we first have to get the list or dictionary object from the database and since it doesn't inherit from Persistent, it won't notify the ZODB of any changes to itself. The only way then to notify the ZODB of such a change is to mark the object as 'dirty', which is done by setting the _p_changed attribute of the class in question to True, so that it knows that it has to save the modified attribute to the database once more:

```
1  class Project(persistent.Persistent):
2      def __init__(self, name, title):
3          self.name = name
4          self.title = title
5          self.tasks = {}
6
7      def addTask(self, name, description):
8          task = Task(name, description)
9          self.tasks[name] = task
10         self._p_changed = 1
```

In the Project class we can see the tasks instance attribute is initialized with a dictionary in line 5. In other words, tasks is a non-persistent mutable attribute. When we create a task and assign it to the dictionary with the task name as the key in line 9, the ZODB is not notified an thus we have to notify it ourselves, setting the _p_changed attribute to True in line 10.

Aborting transactions

This is not done anywhere on the TimeTrax code at this point, but it's important to remember that instead of committing a transaction it is entirely possible that some condition in your code arises that requires you to abort it and roll back all changes since the last transaction. This is easily done by calling transaction.abort() instead of commit, as the following example illustrates:

```
def test_abort(self):
    self.addProject('testabort', 'Abort demo')
    self.addTask('testabort', 'abortme', 'Abort this task')
    self.bookTime('testabort', 'abortme', 1, 'Used time')
    transaction.abort()
```

The TimeTrax code

For reference, here is the complete TimeTrax code as it is at the end of this chapter:

```
import sys
import cmd
import shlex

import persistent
import transaction
import ZODB
import ZODB.FileStorage

class Project(persistent.Persistent):
    def __init__(self, name, title):
        self.name = name
        self.title = title
        self.tasks = {}

    def addTask(self, name, description):
        task = Task(name, description)
        self.tasks[name] = task
        self._p_changed = True

class Task(persistent.Persistent):
    def __init__(self, name,  description):
        self.name = name
        self.description = description
        self.bookings = []

    def bookTime(self, time, description=''):
        booking = Booking(time, description)
        self.bookings.append(booking)
        self._p_changed = True

class Booking(persistent.Persistent):
    def __init__(self, time, description):
        self.time = time
        self.description = description

class TimeTrax(cmd.Cmd, object):
    def __init__(self, intro="TimeTrax time tracking helper",
```

```
                prompt="timetrax: ", db_path="projects.fs"):
        super(TimeTrax, self).__init__()
        self.intro = intro
        self.prompt = prompt
        self.db = ZODB.DB(ZODB.FileStorage.FileStorage(db_path))
        self.projects = self.db.open().root()

    def addProject(self, name, title):
        project = Project(name, title)
        self.projects[name] = project
        transaction.commit()

    def dropProject(self, name):
        del self.projects[name]
        transaction.commit()

    def addTask(self, project, name, description):
        self.projects[project].addTask(name, description)
        transaction.commit()

    def bookTime(self, project, task, time, description):
        self.projects[project].tasks[task].bookTime(time, description)
        transaction.commit()

    def postloop(self):
        print

    def postcmd(self, stop, line):
        if line=='EOF':
            return self.do_EOF(self)
        if not line.startswith('help'):
            print

    def precmd(self, line):
        if not line.startswith('help'):
            print
        return line

    def emptyline(self):
        print

    def help_help(self):
        print "Show this help"

    def do_EOF(self, line):
        "Exit the shell"
        return True

    def do_create(self, line):
        name, title = shlex.split(line)
        self.addProject(name, title)
        print "created project %s" % name

    def help_create(self):
        print "create project_name project_title"
        print "Create a new project with unique name project_name"

    def do_drop(self, line):
```

```
97          if line in self.projects.keys():
98              self.dropProject(line)
99              print "dropped project %s" % line
100         else:
101             print "%s is not a recognized project" % line
102
103     def help_drop(self):
104         print "drop project_name"
105         print "drop a project named project_name from the database"
106
107     def do_list(self, line):
108         if not line:
109             return self.list_projects()
110         args = shlex.split(line)
111         if len(args) == 1:
112             return self.list_tasks(line)
113         return self.list_bookings(args[0], args[1])
114
115     def help_list(self):
116         print "list [project_name] [task_name]"
117         print "List all projects if no arguments are given"
118         print "List all tasks in project_name"
119         print "List all time bookings for task_name in project_name"
120
121     def list_projects(self):
122         print "Project             Title"
123         print "=======             ====="
124         for name, project in self.projects.items():
125             print "%-20s%s" % (name, project.title)
126
127     def list_tasks(self, project):
128         print "Project: %s" % project
129         print
130         print "Task                 Time    Description"
131         print "====                 ====    ==========="
132         for name, task in self.projects[project].tasks.items():
133             total_time = sum([booking.time for booking in task.bookings])
134             print "%-20s%-8s%s" % (name, total_time, task.description)
135
136     def list_bookings(self, project, task):
137         task = self.projects[project].tasks[task]
138         total_time = 0
139         print "Project: %s" % project
140         print "task: %s" % task.description
141         print
142         print "Time    Description"
143         print "====    ==========="
144         for booking in task.bookings:
145             print "%-8s%s" % (booking.time, booking.description)
146             total_time = total_time + booking.time
147         print "----    -----------"
148         print "%-8sTotal" % total_time
149
150     def do_add(self, line):
151         project, task, description = shlex.split(line)
152         self.addTask(project, task, description)
153         print "Added task %s to project %s" % (task, project)
154
```

```
155     def help_add(self):
156         print "add project_name task_name task_description"
157         print "Add a new task to project_name"
158         print "task_description is required"
159 
160     def do_book(self, line):
161         args = shlex.split(line)
162         if len(args) == 3:
163             project, task, time= shlex.split(line)
164             description = ''
165         else:
166             project, task, time, description = shlex.split(line)
167         time = int(time)
168         self.bookTime(project, task, time, description)
169         print "booked %s hours for task %s in project %s" % (time,
170                                                              task,
171                                                              project)
172 
173     def help_book(self):
174         print "book project_name task_name hours work_description"
175         print "Book time in hours for a task in project_name"
176         print "work_description is required"
177 
178 if __name__ == '__main__':
179     if len(sys.argv) > 1:
180         args = [(' ' in arg and '"%s"' % arg) or arg for arg in sys.argv[1:]]
181         TimeTrax().onecmd(' '.join(args))
182     else:
183         TimeTrax().cmdloop()
```

Summary

In this chapter, we have covered the most important aspects of working with the ZODB and developed a fully working command line application for tracking projects. The next chapter will examine in detail a key aspect of the ZODB: the transaction machinery.

CHAPTER 6

Transactions

A key feature of the ZODB is its support for transactions. Changes made to any data stored inside the database are not persisted until the transaction is committed. Obviously, this means that we can also abort or roll back a transaction.

In fact, the transaction mechanism used by the ZODB is much more powerful than that. It offers a two-phase commit protocol which allows multiple database backends, even non-ZODB databases, 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.

The best part is that this transaction mechanism is not tied to the ZODB and can be used in Python applications as a general transaction support library. Because of this and also because understanding the transaction package is important to use the ZODB correctly, this chapter describes the package in detail and shows how to use it outside the ZODB.

Getting the transaction package

To install the transaction package you can use easy_install:

```
$ easy_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.

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 for the transaction package is thread aware. Each thread is associated with a unique transaction.

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

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

Using transactions

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.

Installing SQLAlchemy

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

```
$ easy_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.

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:

```
$ easy_install zope.sqlalchemy
```

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

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:

```
>>> from sqlalchemy import create_engine
>>> 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.

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'
```

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()
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 if course we have to hold on to it in case we do.

Now, we'll add a second user:

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

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

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

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

Managing more than one backend

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

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

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

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

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

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

Now, let's set up a ZODB connection, like we learned in the previous chapters:

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

```

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

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

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

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

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

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

The two-phase commit protocol in practice

Now that we have seen how transactions work in practice, let's take a deeper look at the two-phase commit protocol that we described briefly at the start of this chapter.

The last few examples have used the ZopeTransactionExtension from the zope.sqlalchemy package, so we'll look at parts of its code to illustrate the protocol steps. The complete code can be found at http://svn.zope.org/zope.sqlalchemy/trunk/.

The ZopeTransactionExtension uses SQLAlchemy's SessionExtension mechanism to make sure that after a session has begun an instance of the zope.sqlalchemy data manager joins the current transaction. Once this is accomplished, the SQLAlchemy session can be made to behave according to the two-phase commit protocol. That is, a call to transaction.commit() will make sure to call the zope.sqlalchemy data manager in addition to any other data managers that have joined the transaction.

To be part of the two-phase commit, a data manager needs to implement some specific methods. Some people call this a contract, others call it an interface. The important part is that the transaction manager expects to be able to call the methods, so every data manager should have them. if it intends to participate in the two-phase commit. The contract or interface that the zope.sqlalchemy implements is named IDataManager (I stands for Interface, of course).

We'll now go through each step of the two-phase commit methods in order, as declared by the IDataManager interface. Once the commit begins, the methods are called in the order that they are listed, except for tpc_finish and tpc_abort, which are only called if the transaction succeeds (tpc_finish) or fails (tpc_abort).

abort

Outside of the two-phase commit proper, a transaction can be aborted before the commit is even attempted, in case we come across some error condition that makes it impossible to commit. The abort method is used for aborting a transaction and forgetting all changes, as well as end the participation of a data manager in the current transaction.

The zope.sqlalchemy data manager uses it for closing the SQLAlchemy session too:

```
1 def abort(self, trans):
2     if self.tx is not None:
3         self._finish('aborted')
```

The _finish method called on line 3 is responsible for closing the session and is only called if there's an actual transaction associated with this data manager:

```
1 def _finish(self, final_state):
2     assert self.tx is not None
3     session = self.session
4     del _SESSION_STATE[id(self.session)]
5     self.tx = self.session = None
6     self.state = final_state
7     session.close()
```

As we'll see, the cleanup work done by the _finish method is also used by other two-phase commit steps.

tpc_begin

The two-phase commit is initiated when the commit method is called on the transaction, like we did in many examples above. The tpc_begin method is called at the start of the commit to perform any necessary steps for saving the data.

In the case of SQLAlchemy the very first thing that is needed is to flush the session, so that all work performed is ready to be committed:

```
1 def tpc_begin(self, trans):
2     self.session.flush()
```

commit

This is the step where data managers need to prepare to save the changes and make sure that any conflicts or errors that could occur during the save operation are handled. Changes should be ready but not made permanent, because the transaction could still be aborted if other transaction managers are not able to commit.

The zope.sqlalchemy data manager here just makes sure that some work has been actually performed and if not goes ahead and calls _finish to end the transaction:

```
1 def commit(self, trans):
2     status = _SESSION_STATE[id(self.session)]
3     if status is not STATUS_INVALIDATED:
4         self._finish('no work')
```

tpc_vote

The last chance for a data manager to make sure that the data can be saved is the vote. The way to vote 'no' is to raise an exception here.

The zope.sqlalchemy data manager simply calls prepare on the SQLAlchemy transaction here, which will itself raise an exception if there are any problems:

```
1 def tpc_vote(self, trans):
2     if self.tx is not None:
3         self.tx.prepare()
4         self.state = 'voted'
```

tpc_finish

This method is only called if the manager voted 'yes' (no exceptions raised) during the voting step. This makes the changes permanent and should never fail. Any errors here could leave the database in an inconsistent state. In other words, only do things here that are guaranteed to work or you may have a serious error in your hands.

The zope.sqlalchemy data manager calls the SQLAlchemy transaction commit and then calls _finish to perform some cleanup:

```
def tpc_finish(self, trans):
    if self.tx is not None:
        self.tx.commit()
        self._finish('committed')
```

tpc_abort

This method is only called if the manager voted 'no' by raising an exception during the voting step. It abandons all changes and ends the transaction. Just like with the tpc_finish step, an error here is a serious condition.

The zope.sqlalchemy data manager calls the SQLAlchemy transaction rollback here, then performs the usual cleanup:

```
def tpc_abort(self, trans):
    if self.tx is not None: # we may not have voted, and been aborted already
        self.tx.rollback()
        self._finish('aborted commit')
```

summary

As we showed, the two-phase commit consists on a series of methods that are called by the transaction manager on all participating data managers. Each data manager is responsible for making its respective backend perform the required actions.

More features and things to keep in mind about transactions

We now know the basics about how to use the transaction package to control any number of backends using available data managers. There are some other features that we haven't mentioned and some things to be aware of when using this package. We'll cover a few of them in this section.

Joining a transaction

Both the zope.sqlalchemy and the ZODB packages make their data managers join the current transaction automatically, but this doesn't have to be always the case. If you are writing your own package that uses transaction you will need to explicitly make your data managers join the current transaction. This can be done using the transaction machinery:

```
import transaction
import SomeDataManager
current = transaction.get()
current.join(SomeDataManager())
```

To join the current transaction, you use transaction.get() to get it and then call the join method, passing an instance of your data manager that will be joining that transaction from then on.

Before-commit hooks

In some cases, it may be desirable to execute some code right before a transaction is committed. For example, if an operation needs to be performed on all objects changed during a transaction, it might be better to call it once at commit time instead of every time an object is changed, which could slow things down. A pre-commit hook on the transaction is available for this:

```
def some_operation(*args, **kws):
    print "operating..."
    for arg in args:
        print arg
    for k,v in kws.items():
        print k,v
    print "...done"

import transaction
current = transaction.get()
current.addBeforeCommitHook(some_operation, args=(1,2), kws={'a':1})
```

In this example the hook some_operation will be registered and later called when the commit process is started. You can pass to the hook function any number of positional arguments as a tuple and also key/value pairs as a dictionary.

It's possible to register any number of hooks for a given transaction. They will be called in the order that they were registered. It's also possible to register a new hook from within the hook function itself, but care must be taken not to create an infinite loop doing this.

Note that a registered hook is only active for the transaction in question. If you want a later transaction to use the same hook, it has to be registered again. The getBeforeCommitHooks method of a transaction will return a tuple for each hook, with the registered hook, args and kws in the order in which they would be invoked at commit time.

After-commit hooks

After-commit hooks work in the same way as before-commit hooks, except that they are called after the transaction commit succeeds or fails. The hook function is passed a boolean argument with the result of the commit, with True signifying a successful transaction and False an aborted one.

```
def some_operation(success, *args, **kws):
    if success:
        print "transaction succeeded"
    else:
        print "transaction failed"

import transaction
current = transaction.get()
current.addAfterCommitHook(some_operation, args=(1,2), kws={'a':1})
```

The getAfterCommitHooks method of a transaction will return a tuple for each hook, with the registered hook, args and kws in the order in which they would be invoked after commit time.

Commit hooks are never called for doomed or explicitly aborted transactions.

Synchronizers

A synchronizer is an object that must implement beforeCompletion and afterCompletion methods. It's registered with the transaction manager, which calls beforeCompletion when it starts a top-level two-phase commit and afterCompletion when the transaction is committed or aborted.

```
class MySynch(object):
    @classmethod
    def beforeCompletion(cls, transaction):
        print "Commit started"
    @classmethod
    def afterCompletion(cls, transaction):
        print "Commit finished"

import transaction
transaction.manager.registerSynch(MySynch)
```

Synchronizers have the advantage that they have to be registered only once to participate in all transactions managed by the transaction manager with which they are registered. However, the only argument that is passed to them is the transaction itself.

Dooming a transaction

There are cases where we encounter a problem that requires aborting a transaction, but we still need to run some code after that regardless of the transaction result. For example, in a web application it might be necessary to finish validating all the fields of a form even if the first one does not pass, to get all possible errors for showing to the user at the end of the request.

This is why the transaction package allows us to doom a transaction. A doomed transaction behaves the same way as an active transaction but if an attempt to commit it is made, it raises an error and thus forces an abort.

To doom a transaction we simply call doom on it:

```
>>> import transaction
>>> current = transaction.get()
>>> current.doom()
```

The isDoomed method can be used to find out if a transaction is already doomed:

```
>>> current.isDoomed()
True
```

Context manager support

Instead of calling commit or abort explicitly to define transaction boundaries, it's possible to use the context manager protocol and define the boundaries using the with statement. For example, in our SQLAlchemy examples above, we could have used this code after setting up our session:

```
import transaction
session = Session()
with transaction.manager:
    session.add(User(id=1, name='John', fullname='John Smith', password='123'))
    session.add(User(id=2, name='John', fullname='John Watson', password='123'))
```

We can have as many statements as we like inside the with block. If an exception occurs, the transaction will be aborted at the end. Otherwise, it will be committed. Note that if you doom the transaction inside the context, it will still try to commit which will result in a DoomedTransaction exception.

Take advantage of the notes feature

A transaction has a description that can be set using its note method. This is very useful for logging information about a transaction, which can then be analyzed for errors or to collect statistics about usage. It is considered a good practice to make use of this feature.

The transaction notes have to be handled and saved by the storage in use or they can be logged. If the storage doesn't handle them and they are needed, the application must provide a way to do it.

```
import logging

import transaction

from sqlalchemy import create_engine
from sqlalchemy import Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
from zope.sqlalchemy import ZopeTransactionExtension

logging.basicConfig()
log = logging.getLogger('example')

engine = create_engine('postgresql://postgres@127.0.0.1:5432')
Base = declarative_base()
Base.metadata.create_all(engine)

class User(Base):
    __tablename__ = 'users'
    id = Column(Integer, primary_key=True)
    name = Column(String)
    fullname = Column(String)
    password = Column(String)

Base.metadata.create_all(engine)
Session = sessionmaker(bind=engine, extension=ZopeTransactionExtension())

session = Session()
current = transaction.get()
session.add(User(id=1, name='John', fullname='John Smith', password='123'))
note = "added user John with id 1"
current.note(note)
log.warn(note)
```

This example is very simple and will log the transaction even if it fails, but the intention was to give an idea of how transaction notes work and how they could be used.

Application developers must handle concurrency

Reading through this chapter, the question might have occurred to you about how the transaction package handles concurrent edits to the same information. The answer is it doesn't, the application developer has to take care of that.

The most common type of concurrency problem, is when a transaction can't be committed because another transaction has a lock on the resources to be modified. This and other similar errors are called transient errors and they are the easiest to handle. Simply retrying the transaction one or more times is usually enough to get it committed in this case.

This is so common that the default transaction manager will try to find a method named should_retry on each data manager whenever an error occurs during transaction processing. This method gets the error instance as a parameter and must return True if the transaction should be retried and False otherwise.

For example, here's how the zope.sqlalchemy data manager defines this method:

```
def should_retry(self, error):
    if isinstance(error, ConcurrentModificationError):
        return True
    if isinstance(error, DBAPIError):
        orig = error.orig
        for error_type, test in _retryable_errors:
            if isinstance(orig, error_type):
                if test is None:
                    return True
                if test(orig):
                    return True
```

First, the method checks if the error is an instance of the SQLAlchemy ConcurrentModificationError. If this is the case, odds are that retrying the transaction has a good chance of succeeding, so True is returned.

After that, if the error is some kind of DBAPIError, again as defined by SQLAlchemy, the data manager checks the error against its own list of retryable exceptions. If there's a match, there are two possibilities: if a test function was not defined for the error in question, True is immediately returned. However, if there's a test function defined, the error is passed to it to verify whether it's really retryable or not. Again, if it is, True is returned.

This strategy should be enough to handle a good number of transient errors and can be tailored to whatever backend you are using if you are willing to create your own data manager.

There are other kinds of conflicts that can occur during a transaction that must be caught and handled by the application, but these are usually application-specific and must be planned for and solved by the developer.

Retrying transactions

Since retrying a transaction is the usual solution for transient errors, applications that use the transaction package have to be prepared to do that easily.

A simple for loop with a try: except clause could be enough, but that can get very ugly very quickly. Fortunately, transaction managers provide a helper for this case. Here's an example, which assumes that we have performed the same SQLAlchemy setup that we have used in previous examples:

```
import transaction

session = Session()
current = transaction.get()

for attempt in transaction.manager.attempts():
    with attempt:
        session.add(User(id=1, name='John', fullname='John Smith', password='123'))
        session.add(User(id=2, name='John', fullname='John Watson', password='123'))
```

The attempts method of the transaction manager returns an iterator, which by default will try the transaction three times. It's possible to pass a different number to the attempts call to change that. If a transient error is raised while processing the transaction, it is retried up to the specified number of tries.

The data manager is responsible for raising the correct kind of exception here, which should be a subclass of transaction.interfaces.TransientError.

Avoid long running transactions

We have seen that transient errors are many times the result of locked resources or busy backends. One important lesson to take from this is that avoiding long transactions is a very good idea, because the quicker a transaction is finished, the quicker another one can start, which minimizes retries and reduces the load on the backend. Uncommitted transactions in many backends are stored in memory, so a big number of changes on a single transaction can eat away systems resources very fast.

The developer should look for ways of getting the required work done as fast as possible. For example, if a lot of changes are required at once, the application could use batching to avoid committing the whole bunch in one go.

Writing our own data manager

By now we have enough knowledge about how the transaction package implements transactions to create our first data manager. Let's create a simple manager that uses the Python pickle module for storing pickled data.

We will use a very simple design: the data manager will behave like a dictionary. We will be able to perform basic dictionary operations, like setting the value of a new key or changing an existing one. When we commit the transaction, the dictionary items will be stored in a pickle on the filesystem.

The PickleDataManager

Let's open a new file and name it pickledm.py. The first thing to do is to import a few modules:

```
import os
import pickle
import transaction

```

Nothing surprising here, just what we need to be able to create our class:

```
class PickleDataManager(object):

    transaction_manager = transaction.manager
```

We define a class, which we'll call PickleDataManager and assign the default transaction manager as its transaction manager. Now for the longest method of our data manager, which turns out to be __init__:

```
    def __init__(self, pickle_path='Data.pkl'):
        self.pickle_path = pickle_path
        try:
            data_file = open(self.pickle_path, 'rb')
        except IOError:
            data_file = None
        uncommitted = {}
        if data_file is not None:
            try:
                uncommitted = pickle.load(data_file)
            except EOFError:
                pass
        self.uncommitted = uncommitted
        self.committed = uncommitted.copy()
```

The initialization method accepts an optional pickle_path parameter, which is the path on the filesystem where the pickle file will be stored. For this example we are not going to worry a lot about this. The important thing is that once we have the path, we try to open an existing pickle file in lines 3-6. If it doesn't exists we just assign None.

We will use a dictionary named 'uncommitted' as a work area for our data manager. If no data file existed, it will be an empty dictionary. If there is a data file, we try to open it and assign its value to our work area (lines 8-12).

Any changes that we do to our data will be made on the uncommitted dictionary. Additionally, we'll need another dictionary to keep a copy of the data as it was at the start of the transaction. For this, we copy the uncommitted dictionary into another dictionary, which we'll name 'committed'. Using copy is important to avoid altering the committed values unintentionally.

We ant our data manager to function as a dictionary, so we need to implement at least the basic methods of a dictionary to get it working. The trick is to actually make those methods act on the uncommitted dictionary, so that all the operations that we perform are stored there.

```
1     def __getitem__(self, name):
2         return self.uncommitted[name]
3
4     def __setitem__(self, name, value):
5         self.uncommitted[name] = value
6
7     def __delitem__(self, name):
8         del self.uncommitted[name]
9
10    def keys(self):
11        return self.uncommitted.keys()
12
13    def values(self):
14        return self.uncommitted.values()
15
16    def items(self):
17        return self.uncommitted.items()
18
19    def __repr__(self):
20        return self.uncommitted.__repr__()
```

These are fairly simple methods. Basically, for each method we call the corresponding one on the uncommitted dictionary. Remember this acts as a sort of work area and nothing will be stored until we commit.

Now we are ready for the transaction protocol methods. For starters, if we decide to abort the transaction before initiating commit, we need to go back to the original dictionary values:

```
1     def abort(self, transaction):
2         self.uncommitted = self.committed.copy()
```

This is very easy to do, since we have a copy of the dictionary as it was at the start of the transaction, so we just copy it over.

For the next couple of methods of the two-phase commit protocol, we don't have to do anything for our simple data manager:

```
1     def tpc_begin(self, transaction):
2         pass
3
4     def commit(self, transaction):
5         pass
```

The tpc_begin method can be used to get the data about to be committed out of any buffers or queues in preparation

for the commit, but here we are only using a dictionary, so it's ready to go. The commit method is used to prepare the data for the commit, but there's also nothing we have to do here.

Now comes the time for voting. We want to make sure that the pickle can be created and raise any exceptions here, because the final step of the two-phase commit can't fail.

```
    def tpc_vote(self, transaction):
        devnull = open(os.devnull, 'wb')
        try:
            pickle.dump(self.uncommitted, devnull)
        except (TypeError, pickle.PicklingError):
            raise ValueError("Unpickleable value cannot be saved")
```

We are going to try to dump the pickle to make sure that it will work. We don't care about the result now, just if it can be dumped, so we use devnull for the dump. For simplicity, we just check for pickling errors here. Other error conditions are possible, like a full drive or other disk errors.

Remember, all that the voting method has to do is to raise an error if there is any problem, and the transaction will be aborted in that case. If this happens all that we have to do is to copy the committed value into the work area, so we go back to the starting value.

```
    def tpc_abort(self, transaction):
        self.uncommitted = self.committed.copy()
```

If there were no problems we can now perform the real pickle dump. At this point the data in our work area is officially committed, so we can copy it to the committed dictionary.

```
    def tpc_finish(self, transaction):
        data_file = open(self.pickle_path, 'wb')
        pickle.dump(self.uncommitted, data_file)
        self.committed = self.uncommitted.copy()
```

That's really all there is to it for a basic data manager. Let's add a bit of an advanced feature, though: a savepoint.

```
    def savepoint(self):
        return PickleSavepoint(self)
```

To add savepoint functionality, a data manager needs to have a savepoint method that returns a savepoint object. The savepoint object needs to be able to rollback to the saved state:

```
class PickleSavepoint(object):

    def __init__(self, dm):
        self.dm = dm
        self.saved_committed = self.dm.uncommitted.copy()

    def rollback(self):
        self.dm.uncommitted = self.saved_committed.copy()
```

In the savepoint initialization, we keep a reference to the data manager instance that called the savepoint. We also copy the uncommitted dictionary to another dictionary stored on the savepoint. If the rollback method is ever called, we'll copy this value again directly into the data managers work area, so that it goes back to the state it was in before the savepoint.

One final method that we'll implement here is sortKey. This method needs to return a string value that is used for setting the order of operations when more than one data manager participates in a transaction. The keys are sorted alphabetically and the different data managers' two-phase commit methods are called in the resulting order.

```
    def sortKey(self):
        return 'pickledm' + str(id(self))
```

In this case we just return a string with the 'pickledm' identifier, since it's not important in what order our data manager is called. There are cases when this feature can be very useful. For example, a data manager that does not support rollbacks can try to return a key that is sorted last, so that it commits during tpc_vote only if the other backends in the same transaction that do support rollback have not rolled back at that point.

For easy reference, here's the full source of our data manager:

```
import os
import pickle
import transaction

class PickleDataManager(object):

    transaction_manager = transaction.manager

    def __init__(self, pickle_path='Data.pkl'):
        self.pickle_path = pickle_path
        try:
            data_file = open(self.pickle_path, 'rb')
        except IOError:
            data_file = None
        uncommitted = {}
        if data_file is not None:
            try:
                uncommitted = pickle.load(data_file)
            except EOFError:
                pass
        self.uncommitted = uncommitted
        self.committed = uncommitted.copy()

    def __getitem__(self, name):
        return self.uncommitted[name]

    def __setitem__(self, name, value):
        self.uncommitted[name] = value

    def __delitem__(self, name):
        del self.uncommitted[name]

    def keys(self):
        return self.uncommitted.keys()

    def values(self):
        return self.uncommitted.values()

    def items(self):
        return self.uncommitted.items()

    def __repr__(self):
        return self.uncommitted.__repr__()

    def abort(self, transaction):
        self.uncommitted = self.committed.copy()

    def tpc_begin(self, transaction):
```

```
        pass

    def commit(self, transaction):
        pass

    def tpc_vote(self, transaction):
        devnull = open(os.devnull, 'wb')
        try:
            pickle.dump(self.uncommitted, devnull)
        except (TypeError, pickle.PicklingError):
            raise ValueError("Unpickleable value cannot be saved")

    def tpc_finish(self, transaction):
        data_file = open(self.pickle_path, 'wb')
        pickle.dump(self.uncommitted, data_file)
        self.committed = self.uncommitted.copy()

    def tpc_abort(self, transaction):
        self.uncommitted = self.committed.copy()

    def sortKey(self):
        return 'pickledm' + str(id(self))

    def savepoint(self):
        return PickleSavepoint(self)


class PickleSavepoint(object):

    def __init__(self, dm):
        self.dm = dm
        self.saved_committed = self.dm.uncommitted.copy()

    def rollback(self):
        self.dm.uncommitted = self.saved_committed.copy()

```

Using our data manager

To use our pickle data manager, we just need to instantiate it, make it join a transaction, perform dictionary-like operations with it and commit. Here's a quick example:

```
import transaction
from pickledm import PickleDataManager

dm = PickleDataManager()
t = transaction.get()
t.join(dm)
dm['bar'] = 'foo'
dm['baz'] = ['s', 'p', 'a', 'm']
transaction.commit()
```

Using transactions in web applications

Nowadays many development projects happen on the web and many web applications require integration of multiple systems or platforms. While the majority of applications may still be 100% based on relational database backends, there are more and more cases where it becomes necessary to combine traditional backends with other types of systems. The transaction package can be very useful in some of these projects.

In fact, the Zope web application server, where the ZODB was born, has been doing combined transaction processing of this kind for more than a decade now. Developers who use applications like the Plone Content Management System still take advantage of this functionality today.

For many years, the transaction support in Zope was tightly integrated with the ZODB, so it has seen very little use outside of Zope. The ongoing evolution of the Python packaging tools and in particular the existence of the Python Package Index have influenced many members of the Zope community and this has led to a renewed interest in making useful Zope tools available for the benefit of the lager Python community.

One project which has been fairly successful in promoting the use of important Zope technologies is the Repoze project (http://www.repoze.org). The main objective of this project is to bridge Zope technologies and WSGI, the Python web server gateway standard. Under this banner, several packages have been released to date that allow using some Zope technologies independently of the Zope framework itself.

Some of these packages can be used with the ZODB, so we'll have occasion to work with them later, but the one that we will discuss now will allow us to work with transactions using WSGI.

Repoze.tm2: transaction aware middleware for WSGI applications

WSGI is the dominant way to serve Python web applications these days. WSGI allows connecting applications together using pipelines and this has spawned the development of many middleware packages that wrap an application and perform some service at the beginning and ending of a web request.

One of these packages is repoze.tm2, a middleware from the Repoze project which uses the transaction package to start a new transaction on every request and commit or abort it after the wrapped application finishes its work, depending on if there were any errors or not.

It's not necessary to call commit or abort manually in application code. All that's needed is that there is a data manager associated with every backend that will participate in the transaction and that this data manager joins the transaction explicitly.

To use repoze.tm2, you first need to add it to your WSGI pipeline. If you are using PasteDeploy for deploying your applications, that means that the repoze.tm2 egg needs to be added to your main pipeline in your .ini configuration file:

```
[pipeline:main]
pipeline =
        egg:repoze.tm2#tm
        myapp
```

In this example, we have an app named 'myapp', which is the main application. By adding the repoze.tm2 egg before it, we are assured that a transaction will be started before calling the main app.

The same thing can be accomplished in Python easily:

```
1 from somewhere import myapp
2 from repoze.tm import TM
3
4 wrapped_app = TM(myapp)
```

Once repoze.tm2 is in the pipeline, all that's needed is to join each data manager that we want to use into the transaction:

```
import transaction
import MyDataManager

dm = MyDataManager()
t = transaction.get()
t.join(dm)
```

That's basically all that there's to it. Any exception raised after this will cause the transaction to abort at the end. Otherwise, the transaction will be committed.

Of course, in a web application there may be some conditions which do not result on an exception, yet are bad enough to warrant aborting the transaction. For example, all 404 or 500 responses from the server indicate errors, even if an exception was never raised.

To handle this situation, repoze.tm2 uses the concept of a commit veto. To use it you need to define a callback in your application that returns True if the transaction should be aborted. In that callback you can analyze the environ and request headers and decide if there is information there that makes aborting necessary. To illustrate, let's take a look at the default commit veto callback included with repoze.tm2:

```
def commit_veto(environ, status, headers):
    for header_name, header_value in headers:
        header_name = header_name.lower()
        if header_name == 'x-tm':
            header_value = header_value.lower()
            if header_value == 'commit':
                return False
            return True
    for bad in ('4', '5'):
        if status.startswith(bad):
            return True
    return False
```

As you can see, this commit veto looks for a header named `x-tm` and returns `True` if the header's value is not `commit`; it also returns `True` if there is a 40x or 50x response from the server. When the commit veto returns `True`, the transaction is aborted.

To use your own commit veto you need to configure it into the middleware. On PasteDeploy configurations:

```
[filter:tm]
commit_veto = my.package:commit_veto
```

The same registration using Python:

```
from otherplace import mywsgiapp
from my.package import commit_veto

from repoze.tm import TM
new_wsgiapp = TM(mywsgiapp, commit_veto=commit_veto)
```

To use the default commit veto, simply substitute the mypackage commit_veto with the one from repoze.tm2:

```
from repoze.tm import default_commit_veto
```

Finally, if some code needs to be run at the end of a transaction, there is an after-end registry that lets you register callbacks to be used after the transaction ends. This can be very useful if you need to perform some cleanup at the end, like closing a connection or logging the result of the transaction.

The after-end callback is registered like this:

```
from repoze.tm import after_end
import transaction
t = transaction.get()
def callback():
    pass # do the cleanup actions
after_end.register(callback, t)
```

A to-do application using repoze.tm2

We'll finish up this long introduction to the transaction package with a simple web application to manage a to-do list. We'll use the pickle data manager that we developed earlier in this chapter along with the repoze.tm2 middleware that we just discussed.

We will use the Pyramid web application framework (http://pylonsproject.org). Pyramid is a very flexible framework and it's very easy to get started with it. It also allows us to create "single file" applications, which is very useful in this case, to avoid lengthy setup instructions or configuration.

To use Pyramid, we recommend creating a virtualenv and installing the Pyramid and repoze.tm2 packages there:

```
$ virtualenv --no-site-packages todoapp
$ cd todoapp
$ bin/easy_install pyramid repoze.tm2
```

The transaction package is a dependency as well, but will be pulled automatically by repoze.tm2.

We want to use our pickle data manager too, so copy the pickledm.py file we created earlier to the virtualenv root.

Now we are ready to write our application. Start a file named todo.py. Make sure it's on the virtualenv root too. Add the following imports there:

```
import os
import time
import transaction

from paste.httpserver import serve

from pyramid.view import view_config
from pyramid.config import Configurator

from repoze.tm import TM
from repoze.tm import default_commit_veto

from pickledm import PickleDataManager
```

You will see some old friends here, like transaction and our pickledm module. On line 5 we import the serve method from paste.httpserver, which we will use to serve our application. Lines 7 and 8 import the view configuration machinery of the Pyramid framework and a Configurator object to configure our application. Finally, lines 10 and 11 import the TM wrapper and the commit veto function that we discussed in the previous section.

Since we have no package to hold our application's files, we have to make sure that we can find the page template that we'll use for rendering our app, so we set that up next:

```
here = os.path.dirname(os.path.abspath(__file__))
template = os.path.join(here, 'todo.pt')
```

In Pyramid, you can define a root object, very similar to what you get when you connect to a ZODB database. The root object points to the root of the web site:

```
1 class Root(object):
2     def __init__(self, request):
3         self.request = request
```

The root object idea is part of a way of defining the structure of a site called traversal. Using traversal, instead of configuring application URLs using regular expressions, like many web frameworks, we define a resource tree which starts at this root object and could potentially contain thousands of other branches. In this case, however, one root object is all that we need for our application.

Pyramid allows us to define views as any callable object. In this case, we'll use a class to define our views, because this enables us to use the class' __init__ method as a common setup area for the collection of individual views that we will define.

```
1 class TodoView(object):
2
3     def __init__(self, request):
4         self.request = request
5         self.dm = PickleDataManager()
6         t = transaction.get()
7         t.join(self.dm)
```

See how we instantiate our pickle data manager and make it join the current transaction. All the views defined in this class will have access to our data manager.

Pyramid allows the use of decorators to configure application views. There are several predicates that we can use inside a view configuration. For our simple to-do application we'll define five views: one for the initial page that will be shown when accessing the site and one each for adding, closing, reopening and deleting tasks.

Remember the Root object that we defined above? This is where we finally use it. We are going to define the application's main view and the Root object will be the context of that view. Context basically means the last object in the URL that represents a path to the resource from the root of the resource tree. The context object of a view is available at rendering time and can be used to get resource specific information. In this case, the main view will show all the items that we have stored in our pickle data manager.

In Pyramid, a view must return a Response object, but since it's a very common thing in web development to use the view to pass some values to a template for rendering, there is a renderer predicate in view configuration that lets us give a template path so that Pyramid takes care of the rendering. In that case, returning a dictionary with the values that the template will use is enough for the view.

```
1     @view_config(context=Root, request_method='GET', renderer=template)
2     def todo_view(self):
3         tasks = self.dm.items()
4         tasks.sort()
5         return { 'tasks': tasks, 'status': None }
```

If you take a look at line 1 above, you'll see that we used as a renderer the template that we defined before the class. As we explained above, the context parameter there means the object in the site structure that the view will be applied to. In this case it's the root of the site, though the specific Root object is not actually used in the view code.

The view configuration mechanism in Pyramid is very powerful and makes it easy to assign views which are used or not depending on things like request headers or parameter values. In this case, we use the request method, so that this view will only be called if the method used is GET.

Notice how on line 3 we use the data manager to get all the stored to-do items for showing on the task list.

The next view finally does something transactional. When the request contains the parameter 'add' this view will be

called and a new to-do item will be added to the task list. The renderer is the same template that displays the full task list.

```
    @view_config(context=Root, request_param='add', renderer=template)
    def add_view(self):
        text = self.request.params.get('text')
        key = str(time.time())
        self.dm[key] = {'task_description': text, 'task_completed': False}
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'New task inserted.' }
```

Since this view will only be called when the add button is pressed on the form, we know that there is a parameter on the request with the name 'text'. This is the item that will be added to the task list. In this example application we don't expect any other user than ourselves, so we can safely use the time as a key for the new item value. We assign that key to the data manager, get the updated list of items for sorting and the view is done. Notice that we didn't have to call commit even though there was a change, because repoze.tm2 will do that for us after the request is completed.

The next few views are almost equal to the add view. In the done view we get a list of task ids and mark all of those tasks as completed:

```
    @view_config(context=Root, request_param='done', renderer=template)
    def done_view(self):
        tasks = self.request.params.getall('tasks')
        for task in tasks:
            self.dm[task]['task_completed'] = True
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'Marked tasks as done.' }
```

The done view does exactly the reverse, marking the list of tasks as not completed:

```
    @view_config(context=Root, request_param='not done', renderer=template)
    def not_done_view(self):
        tasks = self.request.params.getall('tasks')
        for task in tasks:
            self.dm[task]['task_completed'] = False
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'Marked tasks as not done.' }

```

Finally, the delete view removes the task with the passed id from our data manager. As with all the other views, there's no need to call commit.

```
    def delete_view(self):
        tasks = self.request.params.getall('tasks')
        for task in tasks:
            del(self.dm[task])
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'Deleted tasks.' }

```

That's really the whole application, all we need now is a way to configure it and start a server process. We'll set this up so that running todo.py with the Python interpreter starts the application:

```
    settings = {}
    config = Configurator(root_factory=Root, settings=settings)
    config.scan()
    app = config.make_wsgi_app()
    app = TM(app, commit_veto=default_commit_veto)
    serve(app, host='0.0.0.0')

```

Pyramid uses a Configurator object to handle application configuration and view registration. On line 2 we create a configurator and then on line 3 we call its scan method to perform the view registration. Be aware that using the decorators to define the views in the code above is not enough for registering them. The scan step is required for doing that.

On line 4 we use the configurator to create a WSGI app and then we wrap that with the repoze.tm2 middleware, to get our automatic transaction commits at the end of each request. We pass in the default_commit_veto as well, so that in the event of 4xx or 5xx response, the transaction is aborted.

Finally, on line 6, we use serve to start serving our application with paste's http server.

We are done, this is the complete source of the application:

```
import os
import time
import transaction

from paste.httpserver import serve

from pyramid.view import view_config
from pyramid.config import Configurator

from repoze.tm import TM
from repoze.tm import default_commit_veto

from pickledm import PickleDataManager

here = os.path.dirname(os.path.abspath(__file__))
template = os.path.join(here, 'todo.pt')

class Root(object):
    def __init__(self, request):
        self.request = request

class TodoView(object):

    def __init__(self, request):
        self.request = request
        self.dm = PickleDataManager()
        t = transaction.get()
        t.join(self.dm)

    @view_config(context=Root, request_method='GET', renderer=template)
    def todo_view(self):
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': None }

    @view_config(context=Root, request_param='add', renderer=template)
    def add_view(self):
        text = self.request.params.get('text')
```

```
        key = str(time.time())
        self.dm[key] = {'task_description': text, 'task_completed': False}
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'New task inserted.' }

    @view_config(context=Root, request_param='done', renderer=template)
    def done_view(self):
        tasks = self.request.params.getall('tasks')
        for task in tasks:
            self.dm[task]['task_completed'] = True
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'Marked tasks as done.' }

    @view_config(context=Root, request_param='not done', renderer=template)
    def not_done_view(self):
        tasks = self.request.params.getall('tasks')
        for task in tasks:
            self.dm[task]['task_completed'] = False
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'Marked tasks as not done.' }

    @view_config(context=Root, request_param='delete', renderer=template)
    def delete_view(self):
        tasks = self.request.params.getall('tasks')
        for task in tasks:
            del(self.dm[task])
        tasks = self.dm.items()
        tasks.sort()
        return { 'tasks': tasks, 'status': 'Deleted tasks.' }

if __name__ == '__main__':
    settings = {}
    config = Configurator(root_factory=Root, settings=settings)
    config.scan()
    app = config.make_wsgi_app()
    app = TM(app, commit_veto=default_commit_veto)
    serve(app, host='0.0.0.0')

```

Our application is almost ready to try, we only need to add a todo.pt template in the same directory as the todo.py file, with the following contents:

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
↪DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" xml:lang="en" xmlns:tal="http://xml.zope.
↪org/namespaces/tal">
<head>
  <title>Todo list demo for repoze.tm2</title>
  <meta http-equiv="Content-Type" content="text/html;charset=UTF-8"/>
  <meta name="keywords" content="python web application" />
  <meta name="description" content="pyramid web application" />
</head>
<body>
  <h1>Todo List</h1>
  <p tal:condition="status"><b>$status</b></p>
```

```
12    <form method="POST">
13      <table tal:condition="tasks">
14        <tr tal:repeat="task tasks">
15          <td>
16            <input type="checkbox" name="tasks"
17              tal:attributes="value task[0]" />
18          </td>
19          <td tal:condition="not task[1]['task_completed']"
20              tal:content="task[1]['task_description']"></td>
21          <td tal:condition="task[1]['task_completed']">
22            <s tal:content="task[1]['task_description']"></s>
23          </td>
24        </tr>
25      </table>
26      <p tal:condition="tasks">
27        <input type="submit" name="done" value="done" />
28        <input type="submit" name="not done" value="not done" />
29        <input type="submit" name="delete" value="delete" />
30      </p>
31      <h3>New task</h3>
32      <textarea name="text" rows="5" cols="80"></textarea><br/>
33      <input type="submit" name="add" value="add" />
34    </form>
35  </body>
36  </html>
```

Pyramid has bindings for various template languages, but comes with chameleon and mako "out of the box". In this case, we used chameleon, but as you can see it's a pretty simple form anyway.

The most important part of the template is the loop that starts on line 14. The tal:repeat attribute on the <tr> tag means that for every task in the tasks variable, the contents of the tag should be repeated. The tasks list comes from the dictionary that was returned by the view, you may remember.

The task list comes from the data manager items and thus each of its elements contains a tuple of id (key) and task. Each task is itself a tuple of description and status. These values are used to populate the form with the task list.

You can now run the application and try it out on the browser. From the root of the virtualenv type:

```
$ bin/python todo.py
serving on 0.0.0.0:8080 view at http://127.0.0.1:8080
```

You can add, remove and complete tasks and if you restart the application you will find the task list is preserved. Try removing the wrapper and see what happens then.

CHAPTER 7

API Reference

Persistent Types

class persistent.**Persistent**
Base class for persistent objects.

Implements *IPersistent*

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

A persistent wrapper for list objects.

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

Id

class **PersistentList**(*initlist=None*)
A persistent wrapper for list objects.

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

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

A persistent wrapper for mapping objects.

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

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

Storages

Single File

class ZODB.FileStorage.**FileStorage**(*file_name*, *create=False*, *read_only=False*, *stop=None*, *quota=None*, *pack_gc=True*, *pack_keep_old=True*, *packer=None*, *blob_dir=None*)

Storage that saves data in a file

cleanup()
Remove all files created by this storage.

lastTid(*oid*)
Return last serialno committed for object oid.

If there is no serialno for this oid – which can only occur if it is a new object – return None.

load(*oid*, *version=''*)
Return pickle data and serial number.

pack(*t*, *referencesf*, *gc=None*)
Copy data from the current database file to a packed file

Non-current records from transactions with time-stamp strings less than packtss are ommitted. As are all undone records.

Also, data back pointers that point before packtss are resolved and the associated data are copied, since the old records are not copied.

undo(*transaction_id*, *transaction*)
Undo a transaction, given by transaction_id.

Do so by writing new data that reverses the action taken by the transaction.

Usually, we can get by with just copying a data pointer, by writing a file position rather than a pickle. Sometimes, we may do conflict resolution, in which case we actually copy new data that results from resolution.

Client/Server Model

class ZEO.ClientStorage.**ClientStorage**(*addr*, *storage='1'*, *cache_size=20971520*, *name=''*, *wait_timeout=None*, *disconnect_poll=None*, *read_only=0*, *read_only_fallback=0*, *blob_dir=None*, *shared_blob_dir=False*, *blob_cache_size=None*, *blob_cache_size_check=10*, *client_label=None*, *cache=None*, *ssl=None*, *ssl_server_hostname=None*, *client=None*, *var=None*, *min_disconnect_poll=1*, *max_disconnect_poll=None*, *wait=True*, *drop_cache_rather_verify=True*, *credentials=None*, *server_sync=False*, *username=None*, *password=None*, *realm=None*, *_client_factory=<class 'ZEO.asyncio.client.ClientThread'>*)

A storage class that is a network client to a remote storage.

This is a faithful implementation of the Storage API.

This class is thread-safe; transactions are serialized in tpc_begin().

close()
Storage API: finalize the storage, releasing external resources.

copyTransactionsFrom(*other*, *verbose=0*)
Copy transactions from another storage.

This is typically used for converting data from one storage to another. *other* must have an .iterator() method.

getName()
Storage API: return the storage name as a string.

The return value consists of two parts: the name as determined by the name and addr argments to the ClientStorage constructor, and the string 'connected' or 'disconnected' in parentheses indicating whether the storage is (currently) connected.

getSize()
Storage API: an approximate size of the database, in bytes.

history(*oid*, *size=1*)
Storage API: return a sequence of HistoryEntry objects.

info(*dict*)
Server callback to update the info dictionary.

invalidateTransaction(*tid*, *oids*)
Server callback: Invalidate objects modified by tid.

isReadOnly()
Storage API: return whether we are in read-only mode.

is_connected(*test=False*)
Return whether the storage is currently connected to a server.

is_read_only()
Storage API: return whether we are in read-only mode.

iterator(*start=None*, *stop=None*)
Return an IStorageTransactionInformation iterator.

loadSerial(*oid*, *serial*)
Storage API: load a historical revision of an object.

new_oid()
Storage API: return a new object identifier.

notify_disconnected()
Internal: notify that the server connection was terminated.

This is called by ConnectionManager when the connection is closed or when certain problems with the connection occur.

pack(*t=None*, *referencesf=None*, *wait=1*, *days=0*)
Storage API: pack the storage.

Deviations from the Storage API: the referencesf argument is ignored; two additional optional arguments wait and days are provided:

wait – a flag indicating whether to wait for the pack to complete; defaults to true.

days – a number of days to subtract from the pack time; defaults to zero.

record_iternext(*next=None*)
Storage API: get the next database record.

This is part of the conversion-support API.

registerDB(*db*)
Storage API: register a database for invalidation messages.

This is called by ZODB.DB (and by some tests).

The storage isn't really ready to use until after this call.

restore(*oid*, *serial*, *data*, *version*, *prev_txn*, *transaction*)
Write data already committed in a separate database.

serialnos(*args*)
Server callback to pass a list of changed (oid, serial) pairs.

store(*oid*, *serial*, *data*, *version*, *txn*)
Storage API: store data for an object.

storeBlob(*oid*, *serial*, *data*, *blobfilename*, *version*, *txn*)
Storage API: store a blob object.

supportsUndo()
Storage API: return whether we support undo.

tpc_abort(*txn*, *timeout=None*)
Storage API: abort a transaction.

(The timeout keyword argument is for tests to wat longer than they normally would.)

tpc_begin(*txn*, *tid=None*, *status=' '*)
Storage API: begin a transaction.

tpc_finish(*txn*, *f=<function <lambda>>*)
Storage API: finish a transaction.

tpc_vote(*txn*)
Storage API: vote on a transaction.

undo(*trans_id*, *txn*)
Storage API: undo a transaction.

This is executed in a transactional context. It has no effect until the transaction is committed. It can be undone itself.

Zope uses this to implement undo unless it is not supported by a storage.

undoInfo(*first=0*, *last=-20*, *specification=None*)
Storage API: return undo information.

undoLog(*first=0*, *last=-20*, *filter=None*)
Storage API: return a sequence of TransactionDescription objects.

The filter argument should be None or left unspecified, since it is impossible to pass the filter function to the server to be executed there. If filter is not None, an empty sequence is returned.

In-Memory for Testing

class ZODB.DemoStorage.**DemoStorage**(*name=None*, *base=None*, *changes=None*, *close_base_on_close=None*, *close_changes_on_close=None*)
A storage that stores changes against a read-only base database

This storage was originally meant to support distribution of application demonstrations with populated read-only databases (on CDROM) and writable in-memory databases.

Demo storages are extemely convenient for testing where setup of a base database can be shared by many tests.

Demo storages are also handy for staging appplications where a read-only snapshot of a production database (often accomplished using a beforestorage) is combined with a changes database implemented with a `FileStorage`. Demo ZODB storage

A demo storage supports demos by allowing a volatile changed database to be layered over a base database.

The base storage must not change.

class **`DemoStorage`**(*name=None*, *base=None*, *changes=None*, *close_base_on_close=None*, *close_changes_on_close=None*)

A storage that stores changes against a read-only base database

This storage was originally meant to support distribution of application demonstrations with populated read-only databases (on CDROM) and writable in-memory databases.

Demo storages are extemely convenient for testing where setup of a base database can be shared by many tests.

Demo storages are also handy for staging appplications where a read-only snapshot of a production database (often accomplished using a beforestorage) is combined with a changes database implemented with a `FileStorage`.

`DemoStorage.`**`load`**(*storage*, *oid*, *version=''*)

Load the most recent revision of an object by calling loadBefore

Starting in ZODB 5, it's no longer necessary for storages to provide a load method.

This function is mainly intended to facilitate transitioning from load to loadBefore. It's mainly useful for tests that are meant to test storages, but do so by calling load on the storages.

This function will likely become unnecessary and be deprecated some time in the future.

`DemoStorage.`**`pop`**()

Close the changes database and return the base.

`DemoStorage.`**`push`**(*changes=None*)

Create a new demo storage using the storage as a base.

The given changes are used as the changes for the returned storage and `False` is passed as `close_base_on_close`.

`DemoStorage.`**`load`**(*storage*, *oid*, *version=''*)

Load the most recent revision of an object by calling loadBefore

Starting in ZODB 5, it's no longer necessary for storages to provide a load method.

This function is mainly intended to facilitate transitioning from load to loadBefore. It's mainly useful for tests that are meant to test storages, but do so by calling load on the storages.

This function will likely become unnecessary and be deprecated some time in the future.

`DemoStorage.`**`pop`**()

Close the changes database and return the base.

`DemoStorage.`**`push`**(*changes=None*)

Create a new demo storage using the storage as a base.

The given changes are used as the changes for the returned storage and `False` is passed as `close_base_on_close`.

class `ZODB.MappingStorage.`**`MappingStorage`**(*name='MappingStorage'*)

In-memory storage implementation

Note that this implementation is somewhat naive and inefficient with regard to locking. Its implementation is primarily meant to be a simple illustration of storage implementation. It's also useful for testing and exploration where scalability and efficiency are unimportant. A simple in-memory mapping-based ZODB storage

This storage provides an example implementation of a fairly full storage without distracting storage details.

class **DataRecord**(*oid*, *tid*, *data*)
Abstract base class for iterator protocol

class ZODB.MappingStorage.**MappingStorage**(*name='MappingStorage'*)
In-memory storage implementation

Note that this implementation is somewhat naive and inefficient with regard to locking. Its implementation is primarily meant to be a simple illustration of storage implementation. It's also useful for testing and exploration where scalability and efficiency are unimportant.

load(*storage*, *oid*, *version=''*)
Load the most recent revision of an object by calling loadBefore

Starting in ZODB 5, it's no longer necessary for storages to provide a load method.

This function is mainly intended to facilitate transitioning from load to loadBefore. It's mainly useful for tests that are meant to test storages, but do so by calling load on the storages.

This function will likely become unnecessary and be deprecated some time in the future.

not_in_transaction()
The storage is not committing a transaction

opened()
The storage is open

MappingStorage.**load**(*storage*, *oid*, *version=''*)
Load the most recent revision of an object by calling loadBefore

Starting in ZODB 5, it's no longer necessary for storages to provide a load method.

This function is mainly intended to facilitate transitioning from load to loadBefore. It's mainly useful for tests that are meant to test storages, but do so by calling load on the storages.

This function will likely become unnecessary and be deprecated some time in the future.

MappingStorage.**not_in_transaction**()
The storage is not committing a transaction

MappingStorage.**opened**()
The storage is open

Connection Pool

class ZODB.**DB**(*storage*, *pool_size=7*, *pool_timeout=2147483648*, *cache_size=400*, *cache_size_bytes=0*, *historical_pool_size=3*, *historical_cache_size=1000*, *historical_cache_size_bytes=0*, *historical_timeout=300*, *database_name='unnamed'*, *databases=None*, *xrefs=True*, *large_record_size=16777216*, ***storage_args*)
The Object Database

The DB class coordinates the activities of multiple database Connection instances. Most of the work is done by the Connections created via the open method.

The DB instance manages a pool of connections. If a connection is closed, it is returned to the pool and its object cache is preserved. A subsequent call to open() will reuse the connection. There is no hard limit on the

pool size. If more than *pool_size* connections are opened, a warning is logged, and if more than twice that many, a critical problem is logged.

The database provides a few methods intended for application code – open, close, undo, and pack – and a large collection of methods for inspecting the database and its connections' caches.

cacheDetail()
Return object counts by class accross all connections.

cacheDetailSize()
Return non-ghost counts sizes for all connections.

cacheExtremeDetail()
Return information about all of the objects in the object caches.

Information includes a connection number, class, object id, reference count and state. The reference count returned excludes references help by ZODB itself.

cacheMinimize()
Minimize cache sizes for all connections

cacheSize()
Return the total count of non-ghost objects in all object caches

close()
Close the database and its underlying storage.

It is important to close the database, because the storage may flush in-memory data structures to disk when it is closed. Leaving the storage open with the process exits can cause the next open to be slow.

What effect does closing the database have on existing connections? Technically, they remain open, but their storage is closed, so they stop behaving usefully. Perhaps close() should also close all the Connections.

connectionDebugInfo()
Get debugging information about connections

This is especially useful to debug connections that seem to be leaking or open too long. Information includes connection info, the connection before setting, and, if a connection is open, the time it was opened. The info is the result of calling *getDebugInfo()* on the connection, and the connection's cache size.

getCacheSize()
Get the configured cache size (objects).

getCacheSizeBytes()
Get the configured cache size in bytes.

getHistoricalCacheSize()
Get the configured historical cache size (objects).

getHistoricalCacheSizeBytes()
Get the configured historical cache size in bytes.

getHistoricalPoolSize()
Get the configured historical pool size

getHistoricalTimeout()
Get the configured historical pool timeout

getName()
Get the storage name

getPoolSize()
Get the configured pool size

getSize()
Get the approximate database size, in bytes

history(*oid*, *size=1*)
Get revision history information for an object.

See `ZODB.interfaces.IStorage.history()`.

klass
alias of `Connection`

lastTransaction()
Get the storage last transaction id.

new_oid()
Return a new oid from the storage.

Kept for backwards compatibility only. New oids should be allocated in a transaction using an open Connection.

objectCount()
Get the approximate object count

open(*transaction_manager=None*, *at=None*, *before=None*)
Return a database Connection for use by application code.

Note that the connection pool is managed as a stack, to increase the likelihood that the connection's stack will include useful objects.

Parameters

- *transaction_manager*: transaction manager to use. None means use the default transaction manager.
- *at*: a datetime.datetime or 8 character transaction id of the time to open the database with a read-only connection. Passing both *at* and *before* raises a ValueError, and passing neither opens a standard writable transaction of the newest state. A timezone-naive datetime.datetime is treated as a UTC value.
- *before*: like *at*, but opens the readonly state before the tid or datetime.

open_then_close_db_when_connection_closes()
Create and return a connection.

When the connection closes, the database will close too.

pack(*t=None*, *days=0*)
Pack the storage, deleting unused object revisions.

A pack is always performed relative to a particular time, by default the current time. All object revisions that are not reachable as of the pack time are deleted from the storage.

The cost of this operation varies by storage, but it is usually an expensive operation.

There are two optional arguments that can be used to set the pack time: t, pack time in seconds since the epcoh, and days, the number of days to subtract from t or from the current time if t is not specified.

setCacheSize(*size*)
Reconfigure the cache size (non-ghost object count)

setCacheSizeBytes(*size*)
Reconfigure the cache total size in bytes

setHistoricalCacheSize(*size*)
Reconfigure the historical cache size (non-ghost object count)

setHistoricalCacheSizeBytes(*size*)
Reconfigure the historical cache total size in bytes

setHistoricalPoolSize(*size*)
Reconfigure the connection historical pool size

setHistoricalTimeout(*timeout*)
Reconfigure the connection historical pool timeout

setPoolSize(*size*)
Reconfigure the connection pool size

storage = storage object
Database storage, implementing **:interface:'~ZODB.interfaces.IStorage'**

supportsUndo()
Return whether the database supports undo.

transaction(*note=None*)
Execute a block of code as a transaction.

If a note is given, it will be added to the transaction's description.

The `transaction` method returns a context manager that can be used with the `with` statement.

undo(*id*, *txn=None*)
Undo a transaction identified by id.

A transaction can be undone if all of the objects involved in the transaction were not modified subsequently, if any modifications can be resolved by conflict resolution, or if subsequent changes resulted in the same object state.

The value of id should be generated by calling undoLog() or undoInfo(). The value of id is not the same as a transaction id used by other methods; it is unique to undo().

Parameters

- *id*: a transaction identifier
- *txn*: transaction context to use for undo(). By default, uses the current transaction.

undoInfo(**args*, ***kw*)
Return a sequence of descriptions for transactions.

See `ZODB.interfaces.IStorageUndoable.undoInfo()`.

undoLog(**args*, ***kw*)
Return a sequence of descriptions for transactions.

See `ZODB.interfaces.IStorageUndoable.undoLog()`.

undoMultiple(*ids*, *txn=None*)
Undo multiple transactions identified by ids.

A transaction can be undone if all of the objects involved in the transaction were not modified subsequently, if any modifications can be resolved by conflict resolution, or if subsequent changes resulted in the same object state.

The values in ids should be generated by calling undoLog() or undoInfo(). The value of ids are not the same as a transaction ids used by other methods; they are unique to undo().

Parameters

- *ids*: a sequence of storage-specific transaction identifiers
- *txn*: transaction context to use for undo(). By default, uses the current transaction.

Connections

class ZODB.Connection.**Connection**(*db*, *cache_size=400*, *before=None*, *cache_size_bytes=0*)
Connection to ZODB for loading and storing objects.

Connections manage object state in collaboration with transaction managers. They're created by calling the *open()* method on *database* objects.

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

add(*obj*)
Add a new object 'obj' to the database and assign it an oid.

cacheGC()
Reduce cache size to target size.

cacheMinimize()
Deactivate all unmodified objects in the cache.

close(*primary=True*)
Close the Connection.

commit(*transaction*)
Commit changes to an object

db()
Returns a handle to the database this connection belongs to.

get(*oid*)
Return the persistent object with oid 'oid'.

getDebugInfo()
Returns a tuple with different items for debugging the connection.

getTransferCounts(*clear=False*)
Returns the number of objects loaded and stored.

get_connection(*database_name*)
Return a Connection for the named database.

isReadOnly()
Returns True if this connection is read only.

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

onCloseCallback(*f*)
Register a callable, f, to be called by close().

open(*transaction_manager=None*, *delegate=True*)
Register odb, the DB that this Connection uses.

This method is called by the DB every time a Connection is opened. Any invalidations received while the Connection was closed will be processed.

If the global module function resetCaches() was called, the cache will be cleared.

Parameters: odb: database that owns the Connection transaction_manager: transaction manager to use. None means

use the default transaction manager.

register for afterCompletion() calls.

register(*obj*)
Register obj with the current transaction manager.

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

obj must be an object loaded from this Connection.

root
Return the database root object.

setDebugInfo(**args*)
Add the given items to the debug information of this connection.

setstate(*obj*)
Load the state for an (ghost) object

sortKey()
Return a consistent sort key for this connection.

sync()
Manually update the view on the database.

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

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

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

transaction_manager = current transaction manager
Transaction manager associated with the connection when it was opened.

Transactions

transaction.**begin**()
Begin a new transaction in the default manager.

transaction.**commit**()
Commit the current transaction in the default manager.

transaction.**abort**()
Abort the current transaction in the default manager.

transaction.**doom**()
Doom the current transaction in the default manager.

transaction.**isDoomed**()
Check if the current transaction in the default manager is doomed.

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

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.

transaction.**get**()
Get the current transaction from the default manager.

transaction.**manager**
Default transaction manager.

Type *ThreadTransactionManager*

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

Implements *ITransactionManager*

abort()
See ITransactionManager.

begin()
See ITransactionManager.

clearSynchs()
See ITransactionManager.

commit()
See ITransactionManager.

doom()
See ITransactionManager.

get()
See ITransactionManager.

isDoomed()
See ITransactionManager.

registerSynch(*synch*)
See ITransactionManager.

registeredSynchs()
See ITransactionManager.

savepoint(*optimistic=False*)
See ITransactionManager.

unregisterSynch(*synch*)
See ITransactionManager.

class transaction.**ThreadTransactionManager**(*explicit=False*)
Thread-aware transaction manager.

Each thread is associated with a unique transaction.

Bases *TransactionManager*, threading.local

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

Implements *ITransaction*

abort()
See ITransaction.

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

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

commit()
See ITransaction.

doom()
See ITransaction.

getAfterCommitHooks()
See ITransaction.

getBeforeCommitHooks()
See ITransaction.

isDoomed()
See ITransaction.

join(*resource*)
See ITransaction.

note(*text*)
See ITransaction.

register(*obj*)
See ITransaction.

savepoint(*optimistic=False*)
See ITransaction.

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

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

Errors

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

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

A savepoint may be invalid because:

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

Interfaces

interface persistent.interfaces.**IPersistent**
Python persistent interface

A persistent object can be in one of several states:

- Unsaved

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

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

- Saved

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

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

- Sticky

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

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

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

- Changed

  The object has been changed.

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

- Ghost

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

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

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

The following state transitions are possible:

- Unsaved -> Saved

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

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

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

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

- Saved -> Sticky

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

- Saved -> Ghost

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

- Sticky -> Saved

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

- Changed -> Saved

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

- Changed -> Ghost

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

- Ghost -> Saved

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

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

About Invalidation, Deactivation and the Sticky & Ghost States

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

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

- call _p_deactivate()
- set _p_changed to None

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

_p_jar
The data manager for the object.

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

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

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

__getstate__()
Get the object data.

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

__setstate__(*state*)
Set the object data.

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

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

May be set by the data manager.

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

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

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

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

_p_invalidate()
Invalidate the object.

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

_p_serial
The object serial number.

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

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

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

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

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

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

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

_p_delattr(*name*)
Delete persistent meta data

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

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

_p_oid
The object id.

It is up to the data manager to assign this.

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

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

_p_state
The object's persistence state token.

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

_p_activate()
Activate the object.

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

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

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

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

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

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

_p_changed
The persistent state of the object.

This is one of:

None – The object is a ghost.

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

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

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

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

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

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

_p_deactivate()
Deactivate the object.

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

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

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

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

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

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

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

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

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

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

_cache
The pickle cache associated with this connection.

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

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

The object must provide the IPersistent interface.

interface ZODB.interfaces.**IConnection**
Connection to ZODB for loading and storing objects.

The Connection object serves as a data manager. The root() method on a Connection returns the root object for the database. This object and all objects reachable from it are associated with the Connection that loaded them. When a transaction commits, it uses the Connection to store modified objects.

Typical use of ZODB is for each thread to have its own Connection and that no thread should have more than one Connection to the same database. A thread is associated with a Connection by loading objects from that Connection. Objects loaded by one thread should not be used by another thread.

A Connection can be frozen to a serial–a transaction id, a single point in history– when it is created. By default, a Connection is not associated with a serial; it uses current data. A Connection frozen to a serial is read-only.

Each Connection provides an isolated, consistent view of the database, by managing independent copies of objects in the database. At transaction boundaries, these copies are updated to reflect the current state of the database.

You should not instantiate this class directly; instead call the open() method of a DB instance.

In many applications, root() is the only method of the Connection that you will need to use.

A Connection instance is not thread-safe. It is designed to support a thread model where each thread has its own transaction. If an application has more than one thread that uses the connection or the transaction the connection is registered with, the application should provide locking.

The Connection manages movement of objects in and out of object storage.

TODO: We should document an intended API for using a Connection via multiple threads.

TODO: We should explain that the Connection has a cache and that multiple calls to get() will return a reference to the same object, provided that one of the earlier objects is still referenced. Object identity is preserved within a connection, but not across connections.

TODO: Mention the database pool.

A database connection always presents a consistent view of the objects in the database, although it may not always present the most current revision of any particular object. Modifications made by concurrent transactions are not visible until the next transaction boundary (abort or commit).

Two options affect consistency. By default, the mvcc and synch options are enabled by default.

If you pass mvcc=False to db.open(), the Connection will never read non-current revisions of an object. Instead it will raise a ReadConflictError to indicate that the current revision is unavailable because it was written after the current transaction began.

The logic for handling modifications assumes that the thread that opened a Connection (called db.open()) is the thread that will use the Connection. If this is not true, you should pass synch=False to db.open(). When the synch option is disabled, some transaction boundaries will be missed by the Connection; in particular, if a transaction does not involve any modifications to objects loaded from the Connection and synch is disabled, the Connection will miss the transaction boundary. Two examples of this behavior are db.undo() and read-only transactions.

Groups of methods:

User Methods: root, get, add, close, db, sync, isReadOnly, cacheGC, cacheFullSweep, cacheMinimize

Experimental Methods: onCloseCallbacks

Database Invalidation Methods: invalidate

Other Methods: exchange, getDebugInfo, setDebugInfo, getTransferCounts

getDebugInfo()
Returns a tuple with different items for debugging the connection.

Debug information can be added to a connection by using setDebugInfo.

readCurrent(*obj*)
Make sure an object being read is current

This is used when applications want to ensure a higher level of consistency for some operations. This should be called when an object is read and the information read is used to write a separate object.

cacheGC()
Reduce cache size to target size.

Call _p_deactivate() on cached objects until the cache size falls under the target size.

getTransferCounts(*clear=False*)
Returns the number of objects loaded and stored.

If clear is True, reset the counters.

get(*oid*)
Return the persistent object with oid 'oid'.

If the object was not in the cache and the object's class is ghostable, then a ghost will be returned. If the object is already in the cache, a reference to the cached object will be returned.

Applications seldom need to call this method, because objects are loaded transparently during attribute lookup.

Parameters: oid: an object id

Raises KeyError if oid does not exist.

> It is possible that an object does not exist as of the current transaction, but existed in the past. It may even exist again in the future, if the transaction that removed it is undone.

Raises ConnectionStateError if the connection is closed.

isReadOnly()
Returns True if the storage for this connection is read only.

db()
Returns a handle to the database this connection belongs to.

sync()
Manually update the view on the database.

This includes aborting the current transaction, getting a fresh and consistent view of the data (synchronizing with the storage if possible) and calling cacheGC() for this connection.

This method was especially useful in ZODB 3.2 to better support read-only connections that were affected by a couple of problems.

setDebugInfo(**items*)
Add the given items to the debug information of this connection.

connections
A mapping from database name to a Connection to that database.

In multi-database use, the Connections of all members of a database collection share the same .connections object.

In single-database use, of course this mapping contains a single entry.

add(*ob*)
Add a new object 'obj' to the database and assign it an oid.

A persistent object is normally added to the database and assigned an oid when it becomes reachable to an object already in the database. In some cases, it is useful to create a new object and use its oid (_p_oid) in a single transaction.

This method assigns a new oid regardless of whether the object is reachable.

The object is added when the transaction commits. The object must implement the IPersistent interface and must not already be associated with a Connection.

Parameters: obj: a Persistent object

Raises TypeError if obj is not a persistent object.

Raises InvalidObjectReference if obj is already associated with another connection.

Raises ConnectionStateError if the connection is closed.

cacheMinimize()
Deactivate all unmodified objects in the cache.

Call _p_deactivate() on each cached object, attempting to turn it into a ghost. It is possible for individual objects to remain active.

get_connection(*database_name*)
Return a Connection for the named database.

This is intended to be called from an open Connection associated with a multi-database. In that case, database_name must be the name of a database within the database collection (probably the name of a different database than is associated with the calling Connection instance, but it's fine to use the name of the calling Connection object's database). A Connection for the named database is returned. If no connection to that database is already open, a new Connection is opened. So long as the multi-database remains open, passing the same name to get_connection() multiple times returns the same Connection object each time.

close()
Close the Connection.

When the Connection is closed, all callbacks registered by onCloseCallback() are invoked and the cache is garbage collected.

A closed Connection should not be used by client code. It can't load or store objects. Objects in the cache are not freed, because Connections are re-used and the cache is expected to be useful to the next client.

root()
Return the database root object.

The root is a persistent.mapping.PersistentMapping.

onCloseCallback(*f*)
Register a callable, f, to be called by close().

f will be called with no arguments before the Connection is closed.

Parameters: f: method that will be called on *close*

interface ZODB.interfaces.**IDatabase**
Extends: *ZODB.interfaces.IStorageWrapper*

ZODB DB.

undoLog(*first*, *last*, *filter=None*)
Return a sequence of descriptions for undoable transactions.

Application code should call undoLog() on a DB instance instead of on the storage directly.

A transaction description is a mapping with at least these keys:

"time": The time, as float seconds since the epoch, when the transaction committed.

"user_name": The text value of the *.user* attribute on that transaction.

"description": The text value of the *.description* attribute on that transaction.

"id'" A bytes uniquely identifying the transaction to the storage. If it's desired to undo this transaction, this is the *transaction_id* to pass to *undo()*.

In addition, if any name+value pairs were added to the transaction by *setExtendedInfo()*, those may be added to the transaction description mapping too (for example, FileStorage's *undoLog()* does this).

filter is a callable, taking one argument. A transaction description mapping is passed to *filter* for each potentially undoable transaction. The sequence returned by *undoLog()* excludes descriptions for which *filter* returns a false value. By default, *filter* always returns a true value.

ZEO note: Arbitrary callables cannot be passed from a ZEO client to a ZEO server, and a ZEO client's implementation of *undoLog()* ignores any *filter* argument that may be passed. ZEO clients should use the related *undoInfo()* method instead (if they want to do filtering).

Now picture a list containing descriptions of all undoable transactions that pass the filter, most recent transaction first (at index 0). The *first* and *last* arguments specify the slice of this (conceptual) list to be returned:

***first*: This is the index of the first transaction description** in the slice. It must be >= 0.

***last*: If >= 0, first:last acts like a Python slice, selecting** the descriptions at indices *first*, first+1, ..., up to but not including index *last*. At most last-first descriptions are in the slice, and *last* should be at least as large as *first* in this case. If *last* is less than 0, then abs(last) is taken to be the maximum number of descriptions in the slice (which still begins at index *first*). When *last* < 0, the same effect could be gotten by passing the positive first-last for *last* instead.

storage
The object that provides storage for the database

This attribute is useful primarily for tests. Normal application code should rarely, if ever, have a need to use this attribute.

undo(*id*, *txn=None*)
Undo a transaction identified by id.

A transaction can be undone if all of the objects involved in the transaction were not modified subsequently, if any modifications can be resolved by conflict resolution, or if subsequent changes resulted in the same object state.

The value of id should be generated by calling undoLog() or undoInfo(). The value of id is not the same as a transaction id used by other methods; it is unique to undo().

id: a storage-specific transaction identifier txn: transaction context to use for undo().

By default, uses the current transaction.

undoInfo(*first=0*, *last=-20*, *specification=None*)
Return a sequence of descriptions for undoable transactions.

This is like *undoLog()*, except for the *specification* argument. If given, *specification* is a dictionary, and *undoInfo()* synthesizes a *filter* function *f* for *undoLog()* such that *f(desc)* returns true for a transaction description mapping *desc* if and only if *desc* maps each key in *specification* to the same value *specification* maps that key to. In other words, only extensions (or supersets) of *specification* match.

ZEO note: *undoInfo()* passes the *specification* argument from a ZEO client to its ZEO server (while a ZEO client ignores any *filter* argument passed to *undoLog()*).

databases
A mapping from database name to DB (database) object.

In multi-database use, all DB members of a database collection share the same .databases object.

In single-database use, of course this mapping contains a single entry.

close()
Close the database and its underlying storage.

It is important to close the database, because the storage may flush in-memory data structures to disk when it is closed. Leaving the storage open with the process exits can cause the next open to be slow.

What effect does closing the database have on existing connections? Technically, they remain open, but their storage is closed, so they stop behaving usefully. Perhaps close() should also close all the Connections.

pack(*t=None*, *days=0*)
Pack the storage, deleting unused object revisions.

A pack is always performed relative to a particular time, by default the current time. All object revisions that are not reachable as of the pack time are deleted from the storage.

The cost of this operation varies by storage, but it is usually an expensive operation.

There are two optional arguments that can be used to set the pack time: t, pack time in seconds since the epcoh, and days, the number of days to subtract from t or from the current time if t is not specified.

open(*transaction_manager=None*, *serial=''*)
Return an IConnection object for use by application code.

transaction_manager: transaction manager to use. None means use the default transaction manager.

serial: the serial (transaction id) of the database to open. An empty string (the default) means to open it to the newest serial. Specifying a serial results in a read-only historical connection.

Note that the connection pool is managed as a stack, to increase the likelihood that the connection's stack will include useful objects.

history(*oid*, *size=1*)
Return a sequence of history information dictionaries.

Up to size objects (including no objects) may be returned.

The information provides a log of the changes made to the object. Data are reported in reverse chronological order.

Each dictionary has the following keys:

time UTC seconds since the epoch (as in time.time) that the object revision was committed.

tid The transaction identifier of the transaction that committed the version.

user_name The text (unicode) user identifier, if any (or an empty string) of the user on whos behalf the revision was committed.

description The text (unicode) transaction description for the transaction that committed the revision.

size The size of the revision data record.

If the transaction had extension items, then these items are also included if they don't conflict with the keys above.

interface `ZODB.interfaces.IStorage`

A storage is responsible for storing and retrieving data of objects.

Consistency and locking

When transactions are committed, a storage assigns monotonically increasing transaction identifiers (tids) to the transactions and to the object versions written by the transactions. ZODB relies on this to decide if data in object caches are up to date and to implement multi-version concurrency control.

There are methods in IStorage and in derived interfaces that provide information about the current revisions (tids) for objects or for the database as a whole. It is critical for the proper working of ZODB that the resulting tids are increasing with respect to the object identifier given or to the databases. That is, if there are 2 results for an object or for the database, R1 and R2, such that R1 is returned before R2, then the tid returned by R2 must be greater than or equal to the tid returned by R1. (When thinking about results for the database, think of these as results for all objects in the database.)

This implies some sort of locking strategy. The key method is tcp_finish, which causes new tids to be generated and also, through the callback passed to it, returns new current tids for the objects stored in a transaction and for the database as a whole.

The IStorage methods affected are lastTransaction, load, store, and tpc_finish. Derived interfaces may introduce additional methods.

tpc_vote(*transaction*)

Provide a storage with an opportunity to veto a transaction

The argument is the same object passed to tpc_begin.

This call raises a StorageTransactionError if the storage isn't participating in two-phase commit or if it is commiting a different transaction.

If a transaction can be committed by a storage, then the method should return. If a transaction cannot be committed, then an exception should be raised. If this method returns without an error, then there must not be an error if tpc_finish or tpc_abort is called subsequently.

The return value can be None or a sequence of a sequence of object ids, as described in IMultiCommitStorage.tpc_vote.

lastTransaction()

Return the id of the last committed transaction.

If no transactions have been committed, return a string of 8 null (0) characters.

isReadOnly()

Test whether a storage allows committing new transactions

For a given storage instance, this method always returns the same value. Read-only-ness is a static property of a storage.

tpc_finish(*transaction*, *func=<function <lambda> at 0x7f151ea03230>*)

Finish the transaction, making any transaction changes permanent.

Changes must be made permanent at this point.

This call raises a StorageTransactionError if the storage isn't participating in two-phase commit or if it is committing a different transaction. Failure of this method is extremely serious.

The first argument is the same object passed to tpc_begin.

The second argument is a call-back function that must be called while the storage transaction lock is held. It takes the new transaction id generated by the transaction.

The return value may be None or the transaction id of the committed transaction, as described in IMultiCommitStorage.

getName()
The name of the storage

The format and interpretation of this name is storage dependent. It could be a file name, a database name, etc..

This is used soley for informational purposes.

loadBefore(*oid*, *tid*)
Load the object data written before a transaction id

If there isn't data before the object before the given transaction, then None is returned, otherwise three values are returned:

- The data record
- The transaction id of the data record
- The transaction id of the following revision, if any, or None.

If the object id isn't in the storage, then POSKeyError is raised.

sortKey()
Sort key used to order distributed transactions

When a transaction involved multiple storages, 2-phase commit operations are applied in sort-key order. This must be unique among storages used in a transaction. Obviously, the storage can't assure this, but it should construct the sort key so it has a reasonable chance of being unique.

The result must be a string.

registerDB(*wrapper*)
Register a storage wrapper IStorageWrapper.

The passed object is a wrapper object that provides an upcall interface to support composition.

Note that, for historical reasons, this is called registerDB rather than register_wrapper.

getSize()
An approximate size of the database, in bytes.

This is used soley for informational purposes.

tpc_abort(*transaction*)
Abort the transaction.

The argument is the same object passed to tpc_begin.

Any changes made by the transaction are discarded.

This call is ignored is the storage is not participating in two-phase commit or if the given transaction is not the same as the transaction the storage is commiting.

new_oid()
Allocate a new object id.

The object id returned is reserved at least as long as the storage is opened.

The return value is a string.

store(*oid*, *serial*, *data*, *version*, *transaction*)
Store data for the object id, oid.

Arguments:

oid The object identifier. This is either a string consisting of 8 nulls or a string previously returned by new_oid.

serial The serial of the data that was read when the object was loaded from the database. If the object was created in the current transaction this will be a string consisting of 8 nulls.

data The data record. This is opaque to the storage.

version This must be an empty string. It exists for backward compatibility.

transaction The object passed to tpc_begin

Several different exceptions may be raised when an error occurs.

ConflictError is raised when serial does not match the most recent serial number for object oid and the conflict was not resolved by the storage.

StorageTransactionError is raised when transaction does not match the current transaction.

StorageError or, more often, a subclass of it is raised when an internal error occurs while the storage is handling the store() call.

tpc_begin(*transaction*)
Begin the two-phase commit process.

The argument provides IStorageTransactionMetaData.

If storage is already participating in a two-phase commit using the same transaction, a StorageTransaction-Error is raised.

If the storage is already participating in a two-phase commit using a different transaction, the call blocks until the current transaction ends (commits or aborts).

close()
Close the storage.

Finalize the storage, releasing any external resources. The storage should not be used after this method is called.

Note that databases close their storages when they're closed, so this method isn't generally called from application code.

loadSerial(*oid*, *serial*)
Load the object record for the give transaction id

If a matching data record can be found, it is returned, otherwise, POSKeyError is raised.

pack(*pack_time*, *referencesf*)
Pack the storage

It is up to the storage to interpret this call, however, the general idea is that the storage free space by:

•discarding object revisions that were old and not current as of the given pack time.

- garbage collecting objects that aren't reachable from the root object via revisions remaining after discarding revisions that were not current as of the pack time.

The pack time is given as a UTC time in seconds since the epoch.

The second argument is a function that should be used to extract object references from database records. This is needed to determine which objects are referenced from object revisions.

__len__()
The approximate number of objects in the storage

This is used soley for informational purposes.

history(*oid*, *size=1*)
Return a sequence of history information dictionaries.

Up to size objects (including no objects) may be returned.

The information provides a log of the changes made to the object. Data are reported in reverse chronological order.

Each dictionary has the following keys:

time UTC seconds since the epoch (as in time.time) that the object revision was committed.

tid The transaction identifier of the transaction that committed the version.

serial An alias for tid, which expected by older clients.

user_name The bytes user identifier, if any (or an empty string) of the user on whos behalf the revision was committed.

description The bytes transaction description for the transaction that committed the revision.

size The size of the revision data record.

If the transaction had extension items, then these items are also included if they don't conflict with the keys above.

interface ZODB.interfaces.**IStorageCurrentRecordIteration**
Extends: *ZODB.interfaces.IStorage*

record_iternext(*next=None*)
Iterate over the records in a storage

Use like this:

```
>>> next = None
>>> while 1:
...     oid, tid, data, next = storage.record_iternext(next)
...     # do things with oid, tid, and data
...     if next is None:
...         break
```

interface ZODB.interfaces.**IStorageWrapper**
Storage wrapper interface

This interface provides 3 facilities:

- Out-of-band invalidation support

A storage can notify it's wrapper of object invalidations that don't occur due to direct operations on the storage. Currently this is only used by ZEO client storages to pass invalidation messages sent from a server.

- Record-reference extraction

  The references method can be used to extract referenced object IDs from a database record. This can be used by storages to provide more advanced garbage collection. A wrapper storage that transforms data will provide a references method that untransforms data passed to it and then pass the data to the layer above it.

- Record transformation

  A storage wrapper may transform data, for example for compression or encryption. Methods are provided to transform or untransform data.

This interface may be implemented by storage adapters or other intermediaries. For example, a storage adapter that provides encryption and/or compresssion will apply record transformations in it's references method.

invalidate(*transaction_id*, *oids*)
Invalidate object ids committed by the given transaction

The oids argument is an iterable of object identifiers.

The version argument is provided for backward compatibility. If passed, it must be an empty string.

transform_record_data(*data*)
Return transformed data

invalidateCache()
Discard all cached data

This can be necessary if there have been major changes to stored data and it is either impractical to enumerate them or there would be so many that it would be inefficient to do so.

references(*record*, *oids=None*)
Scan the given record for object ids

A list of object ids is returned. If a list is passed in, then it will be used and augmented. Otherwise, a new list will be created and returned.

untransform_record_data(*data*)
Return untransformed data

interface ZODB.interfaces.**IStorageIteration**
API for iterating over the contents of a storage.

iterator(*start=None*, *stop=None*)
Return an IStorageTransactionInformation iterator.

If the start argument is not None, then iteration will start with the first transaction whose identifier is greater than or equal to start.

If the stop argument is not None, then iteration will end with the last transaction whose identifier is less than or equal to stop.

The iterator provides access to the data as available at the time when the iterator was retrieved.

interface ZODB.interfaces.**IStorageRecordInformation**
Provide information about a single storage record

data_txn
The previous transaction id, bytes

oid
The object id, bytes

tid
The transaction id, bytes

data
The data record, bytes

interface ZODB.interfaces.**IStorageRestoreable**
Extends: *ZODB.interfaces.IStorage*

Copying Transactions

The IStorageRestoreable interface supports copying already-committed transactions from one storage to another. This is typically done for replication or for moving data from one storage implementation to another.

tpc_begin(*transaction*, *tid=None*)
Begin the two-phase commit process.

If storage is already participating in a two-phase commit using the same transaction, the call is ignored.

If the storage is already participating in a two-phase commit using a different transaction, the call blocks until the current transaction ends (commits or aborts).

The first argument provides IStorageTransactionMetaData.

If a transaction id is given, then the transaction will use the given id rather than generating a new id. This is used when copying already committed transactions from another storage.

restore(*oid*, *serial*, *data*, *version*, *prev_txn*, *transaction*)
Write data already committed in a separate database

The restore method is used when copying data from one database to a replica of the database. It differs from store in that the data have already been committed, so there is no check for conflicts and no new transaction is is used for the data.

Arguments:

oid The object id for the record

serial The transaction identifier that originally committed this object.

data The record data. This will be None if the transaction undid the creation of the object.

prev_txn The identifier of a previous transaction that held the object data. The target storage can sometimes use this as a hint to save space.

transaction The current transaction.

Nothing is returned.

interface ZODB.interfaces.**IStorageTransactionInformation**
Extends: ZODB.interfaces.IStorageTransactionMetaData

Provide information about a storage transaction.

Can be iterated over to retrieve the records modified in the transaction.

Note that this may contain a status field used by FileStorage to support packing. At some point, this will go away when FileStorage has a better pack algoritm.

tid
Transaction id

__iter__()
Iterate over the transaction's records given as IStorageRecordInformation objects.

interface ZODB.interfaces.**IStorageUndoable**
Extends: *ZODB.interfaces.IStorage*

A storage supporting transactional undo.

undoLog(*first*, *last*, *filter=None*)
Return a sequence of descriptions for undoable transactions.

Application code should call undoLog() on a DB instance instead of on the storage directly.

A transaction description is a mapping with at least these keys:

"time": The time, as float seconds since the epoch, when the transaction committed.

"user_name": The bytes value of the *.user* attribute on that transaction.

"description": The bytes value of the *.description* attribute on that transaction.

"id'" A bytes uniquely identifying the transaction to the storage. If it's desired to undo this transaction, this is the *transaction_id* to pass to *undo()*.

In addition, if any name+value pairs were added to the transaction by *setExtendedInfo()*, those may be added to the transaction description mapping too (for example, FileStorage's *undoLog()* does this).

filter is a callable, taking one argument. A transaction description mapping is passed to *filter* for each potentially undoable transaction. The sequence returned by *undoLog()* excludes descriptions for which *filter* returns a false value. By default, *filter* always returns a true value.

ZEO note: Arbitrary callables cannot be passed from a ZEO client to a ZEO server, and a ZEO client's implementation of *undoLog()* ignores any *filter* argument that may be passed. ZEO clients should use the related *undoInfo()* method instead (if they want to do filtering).

Now picture a list containing descriptions of all undoable transactions that pass the filter, most recent transaction first (at index 0). The *first* and *last* arguments specify the slice of this (conceptual) list to be returned:

***first*: This is the index of the first transaction description** in the slice. It must be >= 0.

***last*: If >= 0, first:last acts like a Python slice, selecting** the descriptions at indices *first*, first+1, ..., up to but not including index *last*. At most last-first descriptions are in the slice, and *last* should be at least as large as *first* in this case. If *last* is less than 0, then abs(last) is taken to be the maximum number of descriptions in the slice (which still begins at index *first*). When *last* < 0, the same effect could be gotten by passing the positive first-last for *last* instead.

undoInfo(*first=0*, *last=-20*, *specification=None*)
Return a sequence of descriptions for undoable transactions.

This is like *undoLog()*, except for the *specification* argument. If given, *specification* is a dictionary, and *undoInfo()* synthesizes a *filter* function *f* for *undoLog()* such that *f(desc)* returns true for a transaction description mapping *desc* if and only if *desc* maps each key in *specification* to the same value *specification* maps that key to. In other words, only extensions (or supersets) of *specification* match.

ZEO note: *undoInfo()* passes the *specification* argument from a ZEO client to its ZEO server (while a ZEO client ignores any *filter* argument passed to *undoLog()*).

undo(*transaction_id*, *transaction*)
Undo the transaction corresponding to the given transaction id.

The transaction id is a value returned from undoInfo or undoLog, which may not be a stored transaction identifier as used elsewhere in the storage APIs.

This method must only be called in the first phase of two-phase commit (after tpc_begin but before tpc_vote). It returns a serial (transaction id) and a sequence of object ids for objects affected by the transaction. The serial is ignored and may be None. The return from this method may be None.

supportsUndo()
Return True, indicating that the storage supports undo.

interface `transaction.interfaces.`**IDataManager**

Objects that manage transactional storage.

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

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

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.

sortKey()

Return a key to use for ordering registered DataManagers.

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

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

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.

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.

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.

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.

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.

valid
Boolean indicating whether the savepoint is valid

rollback()
Rollback any work done since the savepoint.

InvalidSavepointRollbackError is raised if the savepoint isn't valid.

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

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

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

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

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

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

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

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

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

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

datamanager must provide the transactions.interfaces.IDataManager interface.

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.

getAfterCommitHooks()
Return iterable producing the registered addAfterCommit hooks.

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

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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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*.

abort()
Abort the transaction.

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

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).

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

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

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

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

commit()
Finalize the transaction.

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

doom()
Doom the transaction.

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

Otherwise the transaction will behave as if it was active.

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

See set_data.

getBeforeCommitHooks()
Return iterable producing the registered addBeforeCommit hooks.

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

interface `transaction.interfaces.`**ITransactionDeprecated**
Deprecated parts of the transaction API.

begin(*info=None*)
Begin a new transaction.

If the transaction is in progress, it is aborted and a new transaction is started using the same transaction object.

register(*object*)
Register the given object for transaction control.

interface `transaction.interfaces.`**ITransactionManager**
An object that manages a sequence of transactions.

Applications use transaction managers to establish transaction boundaries.

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.)

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.

clearSynchs()
Unregister all registered ISynchronizers.

This exists to support test cleanup/initialization

get()
Get the current transaction.

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

unregisterSynch(*synch*)
Unregister an ISynchronizer.

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

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

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

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

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

An ISavepoint object is returned.

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

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()`.

abort()
Abort 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.

registeredSynchs()
Determine if any ISynchronizers are registered.

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

This exists to support test cleanup/initialization

doom()
Doom the current transaction.

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

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

Z