ZODB/ZEO Programming Guide

Release 3.4.0

A.M. Kuchling

June 9, 2005

amk@amk.ca

Contents

---

1 Introduction

This guide explains how to write Python programs that use the Z Object Database (ZODB) and Zope Enterprise Objects (ZEO). The latest version of the guide is always available at http://www.zope.org/Wikis/ZODB/guide/index.html.

1.1 What is the ZODB?

The ZODB is a persistence system for Python objects. Persistent programming languages provide facilities that automatically write objects to disk and read them in again when they’re required by a running program. By installing the ZODB, you add such facilities to Python.

It’s certainly possible to build your own system for making Python objects persistent. The usual starting points are the `pickle` module, for converting objects into a string representation, and various database modules, such as the `gdbm` or `bsddb` modules, that provide ways to write strings to disk and read them back. It’s straightforward to combine the `pickle` module and a database module to store and retrieve objects, and in fact the `shelve` module, included in Python’s standard library, does this.

The downside is that the programmer has to explicitly manage objects, reading an object when it’s needed and writing it out to disk when the object is no longer required. The ZODB manages objects for you, keeping them in a cache, writing them out to disk when they are modified, and dropping them from the cache if they haven’t been used in a while.

1.2 OODBs vs. Relational DBs

Another way to look at it is that the ZODB is a Python-specific object-oriented database (OODB). Commercial object databases for C++ or Java often require that you jump through some hoops, such as using a special preprocessor or avoiding certain data types. As we'll see, the ZODB has some hoops of its own to jump through, but in comparison the naturalness of the ZODB is astonishing.

Relational databases (RDBs) are far more common than OODBs. Relational databases store information in tables; a table consists of any number of rows, each row containing several columns of information. (Rows are more formally called relations, which is where the term "relational database" originates.)

Let's look at a concrete example. The example comes from my day job working for the MEMS Exchange, in a greatly simplified version. The job is to track process runs, which are lists of manufacturing steps to be performed in a semiconductor fab. A run is owned by a particular user, and has a name and assigned ID number. Runs consist of a number of operations; an operation is a single step to be performed, such as depositing something on a wafer or etching something off it.

Operations may have parameters, which are additional information required to perform an operation. For example, if you're depositing something on a wafer, you need to know two things: 1) what you're depositing, and 2) how much should be deposited. You might deposit 100 microns of silicon oxide, or 1 micron of copper.

Mapping these structures to a relational database is straightforward:

```
CREATE TABLE runs (
  int      run_id,
  varchar  owner,
  varchar  title,
  int      acct_num,
  primary key(run_id)
);

CREATE TABLE operations (
  int      run_id,
  int      step_num,
  varchar  process_id,
  PRIMARY KEY(run_id, step_num),
  FOREIGN KEY(run_id) REFERENCES runs(run_id),
);

CREATE TABLE parameters (
  int      run_id,
  int      step_num,
  varchar  param_name,
  varchar  param_value,
  PRIMARY KEY(run_id, step_num, param_name)
  FOREIGN KEY(run_id, step_num)
     REFERENCES operations(run_id, step_num),
);
```

In Python, you would write three classes named `Run`, `Operation`, and `Parameter`. I won't present code for defining these classes, since that code is uninteresting at this point. Each class would contain a single method to begin with, an `__init__` method that assigns default values, such as 0 or `None`, to each attribute of the class.

It's not difficult to write Python code that will create a `Run` instance and populate it with the data from the relational tables; with a little more effort, you can build a straightforward tool, usually called an object-relational mapper, to do this automatically. (See http://www.amk.ca/python/unmaintained/ordb.html for a quick hack at a Python object-relational mapper, and http://www.python.org/workshops/1997-10/proceedings/shprentz.html for Joel Shprentz's more

successful implementation of the same idea; Unlike mine, Shprentz's system has been used for actual work.)

However, it is difficult to make an object-relational mapper reasonably quick; a simple-minded implementation like mine is quite slow because it has to do several queries to access all of an object's data. Higher performance object-relational mappers cache objects to improve performance, only performing SQL queries when they actually need to.

That helps if you want to access run number 123 all of a sudden. But what if you want to find all runs where a step has a parameter named 'thickness' with a value of 2.0? In the relational version, you have two unappealing choices:

1. Write a specialized SQL query for this case: `SELECT run_id FROM operations WHERE param_name = 'thickness' AND param_value = 2.0`

   If such queries are common, you can end up with lots of specialized queries. When the database tables get rearranged, all these queries will need to be modified.

2. An object-relational mapper doesn't help much. Scanning through the runs means that the the mapper will perform the required SQL queries to read run #1, and then a simple Python loop can check whether any of its steps have the parameter you're looking for. Repeat for run #2, 3, and so forth. This does a vast number of SQL queries, and therefore is incredibly slow.

An object database such as ZODB simply stores internal pointers from object to object, so reading in a single object is much faster than doing a bunch of SQL queries and assembling the results. Scanning all runs, therefore, is still inefficient, but not grossly inefficient.

1.3 What is ZEO?

The ZODB comes with a few different classes that implement the `Storage` interface. Such classes handle the job of writing out Python objects to a physical storage medium, which can be a disk file (the `FileStorage` class), a BerkeleyDB file (`BDBFullStorage`), a relational database (`DCOracleStorage`), or some other medium. ZEO adds `ClientStorage`, a new `Storage` that doesn't write to physical media but just forwards all requests across a network to a server. The server, which is running an instance of the `StorageServer` class, simply acts as a front-end for some physical `Storage` class. It's a fairly simple idea, but as we'll see later on in this document, it opens up many possibilities.

1.4 About this guide

The primary author of this guide works on a project which uses the ZODB and ZEO as its primary storage technology. We use the ZODB to store process runs and operations, a catalog of available processes, user information, accounting information, and other data. Part of the goal of writing this document is to make our experience more widely available. A few times we've spent hours or even days trying to figure out a problem, and this guide is an attempt to gather up the knowledge we've gained so that others don't have to make the same mistakes we did while learning.

The author's ZODB project is described in a paper available here, http://www.amk.ca/python/writing/mx-architecture/

This document will always be a work in progress. If you wish to suggest clarifications or additional topics, please send your comments to zodb-dev@zope.org.

1.5 Acknowledgements

Andrew Kuchling wrote the original version of this guide, which provided some of the first ZODB documentation for Python programmers. His initial version has been updated over time by Jeremy Hylton and Tim Peters.

I'd like to thank the people who've pointed out inaccuracies and bugs, offered suggestions on the text, or proposed new topics that should be covered: Jeff Bauer, Willem Broekema, Thomas Guettler, Chris McDonough, George Runyan.

2 ZODB Programming

2.1 Installing ZODB

ZODB is packaged using the standard distutils tools.

Requirements

You will need Python 2.3 or higher. Since the code is packaged using distutils, it is simply a matter of untarring or unzipping the release package, and then running `python setup.py install`.

You'll need a C compiler to build the packages, because there are various C extension modules. Binary installers are provided for Windows users.

Installing the Packages

Download the ZODB tarball containing all the packages for both ZODB and ZEO from http://www.zope.org/Products/ZODB3.3. See the 'README.txt' file in the top level of the release directory for details on building, testing, and installing.

You can find information about ZODB and the most current releases in the ZODB Wiki at http://www.zope.org/Wikis/ZODB.

2.2 How ZODB Works

The ZODB is conceptually simple. Python classes subclass a `persistent.Persistent` class to become ZODB-aware. Instances of persistent objects are brought in from a permanent storage medium, such as a disk file, when the program needs them, and remain cached in RAM. The ZODB traps modifications to objects, so that when a statement such as `obj.size = 1` is executed, the modified object is marked as "dirty." On request, any dirty objects are written out to permanent storage; this is called committing a transaction. Transactions can also be aborted or rolled back, which results in any changes being discarded, dirty objects reverting to their initial state before the transaction began.

The term "transaction" has a specific technical meaning in computer science. It's extremely important that the contents of a database don't get corrupted by software or hardware crashes, and most database software offers protection against such corruption by supporting four useful properties, Atomicity, Consistency, Isolation, and Durability. In computer science jargon these four terms are collectively dubbed the ACID properties, forming an acronym from their names.

The ZODB provides all of the ACID properties. Definitions of the ACID properties are:

Atomicity means that any changes to data made during a transaction are all-or-nothing. Either all the changes are applied, or none of them are. If a program makes a bunch of modifications and then crashes, the database won't be partially modified, potentially leaving the data in an inconsistent state; instead all the changes will be forgotten. That's bad, but it's better than having a partially-applied modification put the database into an inconsistent state.

Consistency means that each transaction executes a valid transformation of the database state. Some databases, but not ZODB, provide a variety of consistency checks in the database or language; for example, a relational database constraint columns to be of particular types and can enforce relations across tables. Viewed more generally, atomicity and isolation make it possible for applications to provide consistency.

Isolation means that two programs or threads running in two different transactions cannot see each other's changes until they commit their transactions.

Durability means that once a transaction has been committed, a subsequent crash will not cause any data to be lost or corrupted.

2.3 Opening a ZODB

There are 3 main interfaces supplied by the ZODB: `Storage`, `DB`, and `Connection` classes. The `DB` and `Connection` interfaces both have single implementations, but there are several different classes that implement the `Storage` interface.

- `Storage` classes are the lowest layer, and handle storing and retrieving objects from some form of long-term storage. A few different types of Storage have been written, such as `FileStorage`, which uses regular disk files, and `BDBFullStorage`, which uses Sleepycat Software's BerkeleyDB database. You could write a new Storage that stored objects in a relational database, for example, if that would better suit your application. Two example storages, `DemoStorage` and `MappingStorage`, are available to use as models if you want to write a new Storage.
- The `DB` class sits on top of a storage, and mediates the interaction between several connections. One `DB` instance is created per process.
- Finally, the `Connection` class caches objects, and moves them into and out of object storage. A multi-threaded program should open a separate `Connection` instance for each thread. Different threads can then modify objects and commit their modifications independently.

Preparing to use a ZODB requires 3 steps: you have to open the `Storage`, then create a `DB` instance that uses the `Storage`, and then get a `Connection` from the `DB instance`. All this is only a few lines of code:

```
from ZODB import FileStorage, DB

storage = FileStorage.FileStorage('/tmp/test-filestorage.fs')
db = DB(storage)
conn = db.open()
```

Note that you can use a completely different data storage mechanism by changing the first line that opens a `Storage`; the above example uses a `FileStorage`. In section 3, "How ZEO Works", you'll see how ZEO uses this flexibility to good effect.

2.4 Using a ZODB Configuration File

ZODB also supports configuration files written in the ZConfig format. A configuration file can be used to separate the configuration logic from the application logic. The storages classes and the `DB` class support a variety of keyword arguments; all these options can be specified in a config file.

The configuration file is simple. The example in the previous section could use the following example:

```
<zodb>
  <filestorage>
  path /tmp/test-filestorage.fs
  </filestorage>
</zodb>
```

The `ZODB.config` module includes several functions for opening database and storages from configuration files.

```
import ZODB.config

db = ZODB.config.databaseFromURL('/tmp/test.conf')
conn = db.open()
```

The ZConfig documentation, included in the ZODB3 release, explains the format in detail. Each configuration file is described by a schema, by convention stored in a 'component.xml' file. ZODB, ZEO, zLOG, and zdaemon all have schemas.

2.5 Writing a Persistent Class

Making a Python class persistent is quite simple; it simply needs to subclass from the `Persistent` class, as shown in this example:

```
from persistent import Persistent

class User(Persistent):
    pass
```

The `Persistent` base class is a new-style class implemented in C.

For simplicity, in the examples the `User` class will simply be used as a holder for a bunch of attributes. Normally the class would define various methods that add functionality, but that has no impact on the ZODB's treatment of the class.

The ZODB uses persistence by reachability; starting from a set of root objects, all the attributes of those objects are made persistent, whether they're simple Python data types or class instances. There's no method to explicitly store objects in a ZODB database; simply assign them as an attribute of an object, or store them in a mapping, that's already in the database. This chain of containment must eventually reach back to the root object of the database.

As an example, we'll create a simple database of users that allows retrieving a `User` object given the user's ID. First, we retrieve the primary root object of the ZODB using the `root()` method of the `Connection` instance. The root object behaves like a Python dictionary, so you can just add a new key/value pair for your application's root object. We'll insert an `OOBTree` object that will contain all the `User` objects. (The `BTree` module is also included as part of Zope.)

```
dbroot = conn.root()

# Ensure that a 'userdb' key is present
# in the root
if not dbroot.has_key('userdb'):
    from BTrees.OOBTree import OOBTree
    dbroot['userdb'] = OOBTree()

userdb = dbroot['userdb']
```

Inserting a new user is simple: create the `User` object, fill it with data, insert it into the `BTree` instance, and commit this transaction.

```
# Create new User instance
import transaction

newuser = User()

# Add whatever attributes you want to track
newuser.id = 'amk'
newuser.first_name = 'Andrew' ; newuser.last_name = 'Kuchling'
...

# Add object to the BTree, keyed on the ID
userdb[newuser.id] = newuser

# Commit the change
transaction.commit()
```

The `transaction` module defines a few top-level functions for working with transactions. `commit()` writes any modified objects to disk, making the changes permanent. `abort()` rolls back any changes that have been made, restoring the original state of the objects. If you're familiar with database transactional semantics, this is all what you'd expect. `get()` returns a `Transaction` object that has additional methods like `note()`, to add a note to the transaction metadata.

More precisely, the `transaction` module exposes an instance of the `ThreadTransactionManager` transaction manager class as `transaction.manager`, and the `transaction` functions `get()` and `begin()` redirect to the same-named methods of `transaction.manager`. The `commit()` and `abort()` functions apply the methods of the same names to the `Transaction` object returned by `transaction.manager.get()`. This is for convenience. It's also possible to create your own transaction manager instances, and to tell `DB.open()` to use your transaction manager instead.

Because the integration with Python is so complete, it's a lot like having transactional semantics for your program's variables, and you can experiment with transactions at the Python interpreter's prompt:

```
>>> newuser
<User instance at 81b1f40>
>>> newuser.first_name           # Print initial value
'Andrew'
>>> newuser.first_name = 'Bob'   # Change first name
>>> newuser.first_name           # Verify the change
'Bob'
>>> transaction.abort()          # Abort transaction
>>> newuser.first_name           # The value has changed back
'Andrew'
```

2.6 Rules for Writing Persistent Classes

Practically all persistent languages impose some restrictions on programming style, warning against constructs they can't handle or adding subtle semantic changes, and the ZODB is no exception. Happily, the ZODB's restrictions are fairly simple to understand, and in practice it isn't too painful to work around them.

The summary of rules is as follows:

- If you modify a mutable object that's the value of an object's attribute, the ZODB can't catch that, and won't mark the object as dirty. The solution is to either set the dirty bit yourself when you modify mutable objects, or use a wrapper for Python's lists and dictionaries (`PersistentList`, `PersistentMapping`) that will set the dirty bit properly.

- Recent versions of the ZODB allow writing a class with `__setattr__`, `__getattr__`, or `__delattr__` methods. (Older versions didn't support this at all.) If you write such a `__setattr__` or `__delattr__` method, its code has to set the dirty bit manually.
- A persistent class should not have a `__del__` method. The database moves objects freely between memory and storage. If an object has not been used in a while, it may be released and its contents loaded from storage the next time it is used. Since the Python interpreter is unaware of persistence, it would call `__del__` each time the object was freed.

Let's look at each of these rules in detail.

Modifying Mutable Objects

The ZODB uses various Python hooks to catch attribute accesses, and can trap most of the ways of modifying an object, but not all of them. If you modify a `User` object by assigning to one of its attributes, as in `userobj.first_name = 'Andrew'`, the ZODB will mark the object as having been changed, and it'll be written out on the following `commit()`.

The most common idiom that *isn't* caught by the ZODB is mutating a list or dictionary. If `User` objects have a attribute named `friends` containing a list, calling `userobj.friends.append(otherUser)` doesn't mark `userobj` as modified; from the ZODB's point of view, `userobj.friends` was only read, and its value, which happened to be an ordinary Python list, was returned. The ZODB isn't aware that the object returned was subsequently modified.

This is one of the few quirks you'll have to remember when using the ZODB; if you modify a mutable attribute of an object in place, you have to manually mark the object as having been modified by setting its dirty bit to true. This is done by setting the `_p_changed` attribute of the object to true:

```
userobj.friends.append(otherUser)
userobj._p_changed = True
```

You can hide the implementation detail of having to mark objects as dirty by designing your class's API to not use direct attribute access; instead, you can use the Java-style approach of accessor methods for everything, and then set the dirty bit within the accessor method. For example, you might forbid accessing the `friends` attribute directly, and add a `get_friend_list()` accessor and an `add_friend()` modifier method to the class. `add_friend()` would then look like this:

```
def add_friend(self, friend):
    self.friends.append(otherUser)
    self._p_changed = True
```

Alternatively, you could use a ZODB-aware list or mapping type that handles the dirty bit for you. The ZODB comes with a `PersistentMapping` class, and I've contributed a `PersistentList` class that's included in my ZODB distribution, and may make it into a future upstream release of Zope.

`__getattr__`, `__delattr__`, and `__setattr__`

ZODB allows persistent classes to have hook methods like `__getattr__` and `__setattr__`. There are four special methods that control attribute access; the rules for each are a little different.

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

The other methods are more delicate. They will override the hooks provided by `Persistent`, so user code must call special methods to invoke those hooks anyway.

The `__getattribute__` method will be called for all attribute access; it overrides the attribute access support inherited from `Persistent`. A user-defined `__getattribute__` must always give the `Persistent` base class a chance to handle special attribute, as well as `__dict__` or `__class__`. The user code should call `_p_getattr`, passing the name of the attribute as the only argument. If it returns True, the user code should call `Persistent`'s `__getattribute__` to get the value. If not, the custom user code can run.

A `__setattr__` hook will also override the `Persistent` `__setattr__` hook. User code must treat it much like `__getattribute__`. The user-defined code must call `_p_setattr` first to all `Persistent` to handle special attributes; `_p_setattr` takes the attribute name and value. If it returns True, `Persistent` handled the attribute. If not, the user code can run. If the user code modifies the object's state, it must assigned to `_p_changed`.

A `__delattr__` hooks must be implemented the same was as a the last two hooks. The user code must call `_p_delattr`, passing the name of the attribute as an argument. If the call returns True, `Persistent` handled the attribute; if not, the user code can run.

`__del__` methods

A `__del__` method is invoked just before the memory occupied by an unreferenced Python object is freed. Because ZODB may materialize, and dematerialize, a given persistent object in memory any number of times, there isn't a meaningful relationship between when a persistent object's `__del__` method gets invoked and any natural aspect of a persistent object's life cycle. For example, it is emphatically not the case that a persistent object's `__del__` method gets invoked only when the object is no longer referenced by other objects in the database. `__del__` is only concerned with reachability from objects in memory.

Worse, a `__del__` method can interfere with the persistence machinery's goals. For example, some number of persistent objects reside in a `Connection`'s memory cache. At various times, to reduce memory burden, objects that haven't been referenced recently are removed from the cache. If a persistent object with a `__del__` method is so removed, and the cache was holding the last memory reference to the object, the object's `__del__` method will be invoked. If the `__del__` method then references any attribute of the object, ZODB needs to load the object from the database again, in order to satisfy the attribute reference. This puts the object back into the cache again: such an object is effectively immortal, occupying space in the memory cache forever, as every attempt to remove it from cache puts it back into the cache. In ZODB versions prior to 3.2.2, this could even cause the cache reduction code to fall into an infinite loop. The infinite loop no longer occurs, but such objects continue to live in the memory cache forever.

Because `__del__` methods don't make good sense for persistent objects, and can create problems, persistent classes should not define `__del__` methods.

2.7 Writing Persistent Classes

Now that we've looked at the basics of programming using the ZODB, we'll turn to some more subtle tasks that are likely to come up for anyone using the ZODB in a production system.

Changing Instance Attributes

Ideally, before making a class persistent you would get its interface right the first time, so that no attributes would ever need to be added, removed, or have their interpretation change over time. It's a worthy goal, but also an impractical one unless you're gifted with perfect knowledge of the future. Such unnatural foresight can't be required of any person, so you therefore have to be prepared to handle such structural changes gracefully. In object-oriented database terminology, this is a schema update. The ZODB doesn't have an actual schema specification, but you're changing the software's expectations of the data contained by an object, so you're implicitly changing the schema.

One way to handle such a change is to write a one-time conversion program that will loop over every single object in

the database and update them to match the new schema. This can be easy if your network of object references is quite structured, making it easy to find all the instances of the class being modified. For example, if all `User` objects can be found inside a single dictionary or BTree, then it would be a simple matter to loop over every `User` instance with a `for` statement. This is more difficult if your object graph is less structured; if `User` objects can be found as attributes of any number of different class instances, then there's no longer any easy way to find them all, short of writing a generalized object traversal function that would walk over every single object in a ZODB, checking each one to see if it's an instance of `User`.

Some OODBs support a feature called extents, which allow quickly finding all the instances of a given class, no matter where they are in the object graph; unfortunately the ZODB doesn't offer extents as a feature.

3 ZEO

3.1 How ZEO Works

The ZODB, as I've described it so far, can only be used within a single Python process (though perhaps with multiple threads). ZEO, Zope Enterprise Objects, extends the ZODB machinery to provide access to objects over a network. The name "Zope Enterprise Objects" is a bit misleading; ZEO can be used to store Python objects and access them in a distributed fashion without Zope ever entering the picture. The combination of ZEO and ZODB is essentially a Python-specific object database.

ZEO consists of about 12,000 lines of Python code, excluding tests. The code is relatively small because it contains only code for a TCP/IP server, and for a new type of Storage, `ClientStorage`. `ClientStorage` simply makes remote procedure calls to the server, which then passes them on a regular `Storage` class such as `FileStorage`. The following diagram lays out the system:

XXX insert diagram here later

Any number of processes can create a `ClientStorage` instance, and any number of threads in each process can be using that instance. `ClientStorage` aggressively caches objects locally, so in order to avoid using stale data the ZEO server sends an invalidation message to all the connected `ClientStorage` instances on every write operation. The invalidation message contains the object ID for each object that's been modified, letting the `ClientStorage` instances delete the old data for the given object from their caches.

This design decision has some consequences you should be aware of. First, while ZEO isn't tied to Zope, it was first written for use with Zope, which stores HTML, images, and program code in the database. As a result, reads from the database are *far* more frequent than writes, and ZEO is therefore better suited for read-intensive applications. If every `ClientStorage` is writing to the database all the time, this will result in a storm of invalidate messages being sent, and this might take up more processing time than the actual database operations themselves. These messages are small and sent in batches, so there would need to be a lot of writes before it became a problem.

On the other hand, for applications that have few writes in comparison to the number of read accesses, this aggressive caching can be a major win. Consider a Slashdot-like discussion forum that divides the load among several Web servers. If news items and postings are represented by objects and accessed through ZEO, then the most heavily accessed objects – the most recent or most popular postings – will very quickly wind up in the caches of the `ClientStorage` instances on the front-end servers. The back-end ZEO server will do relatively little work, only being called upon to return the occasional older posting that's requested, and to send the occasional invalidate message when a new posting is added. The ZEO server isn't going to be contacted for every single request, so its workload will remain manageable.

3.2 Installing ZEO

This section covers how to install the ZEO package, and how to configure and run a ZEO Storage Server on a machine.

Requirements

The ZEO server software is included in ZODB3. As with the rest of ZODB3, you'll need Python 2.3 or higher.

Running a server

The runzeo.py script in the ZEO directory can be used to start a server. Run it with the -h option to see the various values. If you're just experimenting, a good choise is to use `python ZEO/runzeo.py -a /tmp/zeosocket -f /tmp/test.fs` to run ZEO with a Unix domain socket and a `FileStorage`.

3.3 Testing the ZEO Installation

Once a ZEO server is up and running, using it is just like using ZODB with a more conventional disk-based storage; no new programming details are introduced by using a remote server. The only difference is that programs must create a `ClientStorage` instance instead of a `FileStorage` instance. From that point onward, ZODB-based code is happily unaware that objects are being retrieved from a ZEO server, and not from the local disk.

As an example, and to test whether ZEO is working correctly, try running the following lines of code, which will connect to the server, add some bits of data to the root of the ZODB, and commits the transaction:

```
from ZEO import ClientStorage
from ZODB import DB
import transaction

# Change next line to connect to your ZEO server
addr = 'kronos.example.com', 1975
storage = ClientStorage.ClientStorage(addr)
db = DB(storage)
conn = db.open()
root = conn.root()

# Store some things in the root
root['list'] = ['a', 'b', 1.0, 3]
root['dict'] = {'a':1, 'b':4}

# Commit the transaction
transaction.commit()
```

If this code runs properly, then your ZEO server is working correctly.

You can also use a configuration file.

```
<zodb>
    <zeoclient>
    server localhost:9100
    </zeoclient>
</zodb>
```

One nice feature of the configuration file is that you don't need to specify imports for a specific storage. That makes the code a little shorter and allows you to change storages without changing the code.

```
import ZODB.config

db = ZODB.config.databaseFromURL('/tmp/zeo.conf')
```

3.4 ZEO Programming Notes

ZEO is written using `asyncore`, from the Python standard library. It assumes that some part of the user application is running an `asyncore` mainloop. For example, Zope run the loop in a separate thread and ZEO uses that. If your application does not have a mainloop, ZEO will not process incoming invalidation messages until you make some call into ZEO. The `Connection.sync` method can be used to process pending invalidation messages. You can call it when you want to make sure the `Connection` has the most recent version of every object, but you don't have any other work for ZEO to do.

3.5 Sample Application: chatter.py

For an example application, we'll build a little chat application. What's interesting is that none of the application's code deals with network programming at all; instead, an object will hold chat messages, and be magically shared between all the clients through ZEO. I won't present the complete script here; it's included in my ZODB distribution, and you can download it from http://www.amk.ca/zodb/demos/. Only the interesting portions of the code will be covered here.

The basic data structure is the `ChatSession` object, which provides an `add_message()` method that adds a message, and a `new_messages()` method that returns a list of new messages that have accumulated since the last call to `new_messages()`. Internally, `ChatSession` maintains a B-tree that uses the time as the key, and stores the message as the corresponding value.

The constructor for `ChatSession` is pretty simple; it simply creates an attribute containing a B-tree:

```
class ChatSession(Persistent):
    def __init__(self, name):
        self.name = name
        # Internal attribute: _messages holds all the chat messages.
        self._messages = BTrees.OOBTree.OOBTree()
```

`add_message()` has to add a message to the `_messages` B-tree. A complication is that it's possible that some other client is trying to add a message at the same time; when this happens, the client that commits first wins, and the second client will get a `ConflictError` exception when it tries to commit. For this application, `ConflictError` isn't serious but simply means that the operation has to be retried; other applications might treat it as a fatal error. The code uses `try...except...else` inside a `while` loop, breaking out of the loop when the commit works without raising an exception.

```
    def add_message(self, message):
        """Add a message to the channel.
        message -- text of the message to be added
        """

        while 1:
            try:
                now = time.time()
                self._messages[now] = message
                get_transaction().commit()
            except ConflictError:
                # Conflict occurred; this process should pause and
                # wait for a little bit, then try again.
                time.sleep(.2)
                pass
            else:
                # No ConflictError exception raised, so break
                # out of the enclosing while loop.
                break
        # end while
```

`new_messages()` introduces the use of *volatile* attributes. Attributes of a persistent object that begin with `_v_` are considered volatile and are never stored in the database. `new_messages()` needs to store the last time the method was called, but if the time was stored as a regular attribute, its value would be committed to the database and shared with all the other clients. `new_messages()` would then return the new messages accumulated since any other client called `new_messages()`, which isn't what we want.

```
    def new_messages(self):
        "Return new messages."

        # self._v_last_time is the time of the most recent message
        # returned to the user of this class.
        if not hasattr(self, '_v_last_time'):
            self._v_last_time = 0

        new = []
        T = self._v_last_time

        for T2, message in self._messages.items():
            if T2 > T:
                new.append(message)
                self._v_last_time = T2

        return new
```

This application is interesting because it uses ZEO to easily share a data structure; ZEO and ZODB are being used for their networking ability, not primarily for their data storage ability. I can foresee many interesting applications using ZEO in this way:

- With a Tkinter front-end, and a cleverer, more scalable data structure, you could build a shared whiteboard using the same technique.
- A shared chessboard object would make writing a networked chess game easy.
- You could create a Python class containing a CD's title and track information. To make a CD database, a read-only ZEO server could be opened to the world, or an HTTP or XML-RPC interface could be written on top of

the ZODB.

- A program like Quicken could use a ZODB on the local disk to store its data. This avoids the need to write and maintain specialized I/O code that reads in your objects and writes them out; instead you can concentrate on the problem domain, writing objects that represent cheques, stock portfolios, or whatever.

4 Transactions and Versioning

4.1 Committing and Aborting

Changes made during a transaction don't appear in the database until the transaction commits. This is done by calling the `commit()` method of the current `Transaction` object, where the latter is obtained from the `get()` method of the current transaction manager. If the default thread transaction manager is being used, then `transaction.commit()` suffices.

Similarly, a transaction can be explicitly aborted (all changes within the transaction thrown away) by invoking the `abort()` method of the current `Transaction` object, or simply `transaction.abort()` if using the default thread transaction manager.

Prior to ZODB 3.3, if a commit failed (meaning the `commit()` call raised an exception), the transaction was implicitly aborted and a new transaction was implicitly started. This could be very surprising if the exception was suppressed, and especially if the failing commit was one in a sequence of subtransaction commits.

So, starting with ZODB 3.3, if a commit fails, all further attempts to commit, join, or register with the transaction raise `ZODB.POSException.TransactionFailedError`. You must explicitly start a new transaction then, either by calling the `abort()` method of the current transaction, or by calling the `begin()` method of the current transaction's transaction manager.

4.2 Subtransactions

Subtransactions can be created within a transaction. Each subtransaction can be individually committed and aborted, but the changes within a subtransaction are not truly committed until the containing transaction is committed.

The primary purpose of subtransactions is to decrease the memory usage of transactions that touch a very large number of objects. Consider a transaction during which 200,000 objects are modified. All the objects that are modified in a single transaction have to remain in memory until the transaction is committed, because the ZODB can't discard them from the object cache. This can potentially make the memory usage quite large. With subtransactions, a commit can be be performed at intervals, say, every 10,000 objects. Those 10,000 objects are then written to permanent storage and can be purged from the cache to free more space.

To commit a subtransaction instead of a full transaction, pass a true value to the `commit()` or `abort()` method of the `Transaction` object.

```
# Commit a subtransaction
transaction.commit(True)

# Abort a subtransaction
transaction.abort(True)
```

A new subtransaction is automatically started upon successful committing or aborting the previous subtransaction.

4.3 Undoing Changes

Some types of `Storage` support undoing a transaction even after it's been committed. You can tell if this is the case by calling the `supportsUndo()` method of the `DB` instance, which returns true if the underlying storage supports undo. Alternatively you can call the `supportsUndo()` method on the underlying storage instance.

If a database supports undo, then the `undoLog(`*start*`, `*end*`[, func])` method on the `DB` instance returns the log of past transactions, returning transactions between the times *start* and *end*, measured in seconds from the epoch. If present, *func* is a function that acts as a filter on the transactions to be returned; it's passed a dictionary representing each transaction, and only transactions for which *func* returns true will be included in the list of transactions returned to the caller of `undoLog()`. The dictionary contains keys for various properties of the transaction. The most important keys are '`id`', for the transaction ID, and '`time`', for the time at which the transaction was committed.

```
>>> print storage.undoLog(0, sys.maxint)
[{'description': '',
  'id': 'AzpGEGqU/0QAAAAAAAAGMA',
  'time': 981126744.98,
  'user_name': ''},
 {'description': '',
  'id': 'AzpGC/hUOKoAAAAAAAAFDQ',
  'time': 981126478.202,
  'user_name': ''}
  ...
```

To store a description and a user name on a commit, get the current transaction and call the `note(`*text*`)` method to store a description, and the `setUser(`*user_name*`)` method to store the user name. While `setUser()` overwrites the current user name and replaces it with the new value, the `note()` method always adds the text to the transaction's description, so it can be called several times to log several different changes made in the course of a single transaction.

```
transaction.get().setUser('amk')
transaction.get().note('Change ownership')
```

To undo a transaction, call the `DB.undo(`*id*`)` method, passing it the ID of the transaction to undo. If the transaction can't be undone, a `ZODB.POSException.UndoError` exception will be raised, with the message "non-undoable transaction". Usually this will happen because later transactions modified the objects affected by the transaction you're trying to undo.

After you call `undo()` you must commit the transaction for the undo to actually be applied. [1] There is one glitch in the undo process. The thread that calls undo may not see the changes to the object until it calls `Connection.sync()` or commits another transaction.

4.4 Versions

Warning: Versions should be avoided. They're going to be deprecated, replaced by better approaches to long-running transactions.

While many subtransactions can be contained within a single regular transaction, it's also possible to contain many regular transactions within a long-running transaction, called a version in ZODB terminology. Inside a version, any number of transactions can be created and committed or rolled back, but the changes within a version are not made

[1] There are actually two different ways a storage can implement the undo feature. Most of the storages that ship with ZODB use the transactional form of undo described in the main text. Some storages may use a non-transactional undo makes changes visible immediately.

visible to other connections to the same ZODB.

Not all storages support versions, but you can test for versioning ability by calling `supportsVersions()` method of the `DB` instance, which returns true if the underlying storage supports versioning.

A version can be selected when creating the `Connection` instance using the `DB.open(`[*version*]`)` method. The *version* argument must be a string that will be used as the name of the version.

```
vers_conn = db.open(version='Working version')
```

Transactions can then be committed and aborted using this versioned connection. Other connections that don't specify a version, or provide a different version name, will not see changes committed within the version named '`Working version`'. To commit or abort a version, which will either make the changes visible to all clients or roll them back, call the `DB.commitVersion()` or `DB.abortVersion()` methods. XXX what are the source and dest arguments for?

The ZODB makes no attempt to reconcile changes between different versions. Instead, the first version which modifies an object will gain a lock on that object. Attempting to modify the object from a different version or from an unversioned connection will cause a `ZODB.POSException.VersionLockError` to be raised:

```
from ZODB.POSException import VersionLockError

try:
    transaction.commit()
except VersionLockError, (obj_id, version):
    print ('Cannot commit; object %s '
           'locked by version %s' % (obj_id, version))
```

The exception provides the ID of the locked object, and the name of the version having a lock on it.

4.5 Multithreaded ZODB Programs

ZODB databases can be accessed from multithreaded Python programs. The `Storage` and `DB` instances can be shared among several threads, as long as individual `Connection` instances are created for each thread.

5 Related Modules

The ZODB package includes a number of related modules that provide useful data types such as BTrees.

5.1 `persistent.mapping.PersistentMapping`

The `PersistentMapping` class is a wrapper for mapping objects that will set the dirty bit when the mapping is modified by setting or deleting a key.

`PersistentMapping`(*container* = {})
: Create a `PersistentMapping` object that wraps the mapping object *container*. If you don't specify a value for *container*, a regular Python dictionary is used.

`PersistentMapping` objects support all the same methods as Python dictionaries do.

5.2 persistent.list.PersistentList

The `PersistentList` class is a wrapper for mutable sequence objects, much as `PersistentMapping` is a wrapper for mappings.

`PersistentList`(*initlist = []*)

Create a `PersistentList` object that wraps the mutable sequence object *initlist*. If you don't specify a value for *initlist*, a regular Python list is used.

`PersistentList` objects support all the same methods as Python lists do.

5.3 BTrees Package

When programming with the ZODB, Python dictionaries aren't always what you need. The most important case is where you want to store a very large mapping. When a Python dictionary is accessed in a ZODB, the whole dictionary has to be unpickled and brought into memory. If you're storing something very large, such as a 100,000-entry user database, unpickling such a large object will be slow. BTrees are a balanced tree data structure that behave like a mapping but distribute keys throughout a number of tree nodes. The nodes are stored in sorted order (this has important consequences – see below). Nodes are then only unpickled and brought into memory as they're accessed, so the entire tree doesn't have to occupy memory (unless you really are touching every single key).

The BTrees package provides a large collection of related data structures. There are variants of the data structures specialized to integers, which are faster and use less memory. There are five modules that handle the different variants. The first two letters of the module name specify the types of the keys and values in mappings – O for any object, I for 32-bit signed integer, and (new in ZODB 3.4) F for 32-bit C float. For example, the `BTrees.IOBTree` module provides a mapping with integer keys and arbitrary objects as values.

The four data structures provide by each module are a BTree, a Bucket, a TreeSet, and a Set. The BTree and Bucket types are mappings and support all the usual mapping methods, e.g. `update()` and `keys()`. The TreeSet and Set types are similar to mappings but they have no values; they support the methods that make sense for a mapping with no keys, e.g. `keys()` but not `items()`. The Bucket and Set types are the individual building blocks for BTrees and TreeSets, respectively. A Bucket or Set can be used when you are sure that it will have few elements. If the data structure will grow large, you should use a BTree or TreeSet. Like Python lists, Buckets and Sets are allocated in one contiguous piece, and insertions and deletions can take time proportional to the number of existing elements. Also like Python lists, a Bucket or Set is a single object, and is pickled and unpickled in its entirety. BTrees and TreeSets are multi-level tree structures with much better (logarithmic) worst-case time bounds, and the tree structure is built out of multiple objects, which ZODB can load individually as needed.

The five modules are named `OOBTree`, `IOBTree`, `OIBTree`, `IIBTree`, and (new in ZODB 3.4) `IFBTree`. The two letter prefixes are repeated in the data types names. The `BTrees.OOBTree` module defines the following types: `OOBTree`, `OOBucket`, `OOSet`, and `OOTreeSet`. Similarly, the other four modules each define their own variants of those four types.

The `keys()`, `values()`, and `items()` methods on BTree and TreeSet types do not materialize a list with all of the data. Instead, they return lazy sequences that fetch data from the BTree as needed. They also support optional arguments to specify the minimum and maximum values to return, often called "range searching". Because all these types are stored in sorted order, range searching is very efficient.

The `keys()`, `values()`, and `items()` methods on Bucket and Set types do return lists with all the data. Starting in ZODB 3.3, there are also `iterkeys()`, `itervalues()`, and `iteritems()` methods that return iterators (in the Python 2.2 sense). Those methods also apply to BTree and TreeSet objects.

A BTree object supports all the methods you would expect of a mapping, with a few extensions that exploit the fact that the keys are sorted. The example below demonstrates how some of the methods work. The extra methods are `minKey()` and `maxKey()`, which find the minimum and maximum key value subject to an optional bound argument, and `byValue()`, which should probably be ignored (it's hard to explain exactly what it does, and as a result it's almost never used – best to consider it deprecated). The various methods for enumerating keys, values and

items also accept minimum and maximum key arguments (”range search”), and (new in ZODB 3.3) optional Boolean arguments to control whether a range search is inclusive or exclusive of the range’s endpoints.

```
>>> from BTrees.OOBTree import OOBTree
>>> t = OOBTree()
>>> t.update({1: "red", 2: "green", 3: "blue", 4: "spades"})
>>> len(t)
4
>>> t[2]
’green’
>>> s = t.keys() # this is a "lazy" sequence object
>>> s
<OOBTreeItems object at 0x0088AD20>
>>> len(s)  # it acts like a Python list
4
>>> s[-2]
3
>>> list(s) # materialize the full list
[1, 2, 3, 4]
>>> list(t.values())
[’red’, ’green’, ’blue’, ’spades’]
>>> list(t.values(1, 2)) # values at keys in 1 to 2 inclusive
[’red’, ’green’]
>>> list(t.values(2))    # values at keys >= 2
[’green’, ’blue’, ’spades’]
>>> list(t.values(min=1, max=4))  # keyword args new in ZODB 3.3
[’red’, ’green’, ’blue’, ’spades’]
>>> list(t.values(min=1, max=4, excludemin=True, excludemax=True))
[’green’, ’blue’]
>>> t.minKey()     # smallest key
1
>>> t.minKey(1.5)  # smallest key >= 1.5
2
>>> for k in t.keys():
...     print k,
1 2 3 4
>>> for k in t:    # new in ZODB 3.3
...     print k,
1 2 3 4
>>> for pair in t.iteritems():  # new in ZODB 3.3
...     print pair,
...
(1, ’red’) (2, ’green’) (3, ’blue’) (4, ’spades’)
>>> t.has_key(4)  # returns a true value, but exactly what undefined
2
>>> t.has_key(5)
0
>>> 4 in t  # new in ZODB 3.3
True
>>> 5 in t  # new in ZODB 3.3
False
>>>
```

Each of the modules also defines some functions that operate on BTrees – `difference()`, `union()`, and `intersection()`. The `difference()` function returns a Bucket, while the other two methods return a Set. If the keys are integers, then the module also defines `multiunion()`. If the values are integers or floats, then the module also defines `weightedIntersection()` and `weightedUnion()`. The function doc strings describe

each function briefly.

`BTrees/Interfaces.py` defines the operations, and is the official documentation. Note that the interfaces don't define the concrete types returned by most operations, and you shouldn't rely on the concrete types that happen to be returned: stick to operations guaranteed by the interface. In particular, note that the interfaces don't specify anything about comparison behavior, and so nothing about it is guaranteed. In ZODB 3.3, for example, two BTrees happen to use Python's default object comparison, which amounts to comparing the (arbitrary but fixed) memory addresses of the BTrees. This may or may not be true in future releases. If the interfaces don't specify a behavior, then whether that behavior appears to work, and exactly happens if it does appear to work, are undefined and should not be relied on.

Total Ordering and Persistence

The BTree-based data structures differ from Python dicts in several fundamental ways. One of the most important is that while dicts require that keys support hash codes and equality comparison, the BTree-based structures don't use hash codes and require a total ordering on keys.

Total ordering means three things:

1. Reflexive. For each *x*, *x* `==` *x* is true.
2. Trichotomy. For each *x* and *y*, exactly one of *x* `<` *y*, *x* `==` *y*, and *x* `>` *y* is true.
3. Transitivity. Whenever *x* `<=` *y* and *y* `<=` *z*, it's also true that *x* `<=` *z*.

The default comparison functions for most objects that come with Python satisfy these rules, with some crucial cautions explained later. Complex numbers are an example of an object whose default comparison function does not satisfy these rules: complex numbers only support `==` and `!=` comparisons, and raise an exception if you try to compare them in any other way. They don't satisfy the trichotomy rule, and must not be used as keys in BTree-based data structures (although note that complex numbers can be used as keys in Python dicts, which do not require a total ordering).

Examples of objects that are wholly safe to use as keys in BTree-based structures include ints, longs, floats, 8-bit strings, Unicode strings, and tuples composed (possibly recursively) of objects of wholly safe types.

It's important to realize that even if two types satisfy the rules on their own, mixing objects of those types may not. For example, 8-bit strings and Unicode strings both supply total orderings, but mixing the two loses trichotomy; e.g., `'x' < chr(255)` and `u'x' == 'x'`, but trying to compare `chr(255)` to `u'x'` raises an exception. Partly for this reason (another is given later), it can be dangerous to use keys with multiple types in a single BTree-based structure. Don't try to do that, and you don't have to worry about it.

Another potential problem is mutability: when a key is inserted in a BTree-based structure, it must retain the same order relative to the other keys over time. This is easy to run afoul of if you use mutable objects as keys. For example, lists supply a total ordering, and then

```
>>> L1, L2, L3 = [1], [2], [3]
>>> from BTrees.OOBTree import OOSet
>>> s = OOSet((L2, L3, L1))  # this is fine, so far
>>> list(s.keys())           # note that the lists are in sorted order
[[1], [2], [3]]
>>> s.has_key([3])           # and [3] is in the set
1
>>> L2[0] = 5                # horrible -- the set is insane now
>>> s.has_key([3])           # for example, it's insane this way
0
>>> s
OOSet([[1], [5], [3]])
>>>
```

Key lookup relies on that the keys remain in sorted order (an efficient form of binary search is used). By mutating key L2 after inserting it, we destroyed the invariant that the OOSet is sorted. As a result, all future operations on this set are unpredictable.

A subtler variant of this problem arises due to persistence: by default, Python does several kinds of comparison by comparing the memory addresses of two objects. Because Python never moves an object in memory, this does supply a usable (albeit arbitrary) total ordering across the life of a program run (an object's memory address doesn't change). But if objects compared in this way are used as keys of a BTree-based structure that's stored in a database, when the objects are loaded from the database again they will almost certainly wind up at different memory addresses. There's no guarantee then that if key K1 had a memory address smaller than the memory address of key K2 at the time K1 and K2 were inserted in a BTree, K1's address will also be smaller than K2's when that BTree is loaded from a database later. The result will be an insane BTree, where various operations do and don't work as expected, seemingly at random.

Now each of the types identified above as "wholly safe to use" never compares two instances of that type by memory address, so there's nothing to worry about here if you use keys of those types. The most common mistake is to use keys that are instances of a user-defined class that doesn't supply its own `__cmp__()` method. Python compares such instances by memory address. This is fine if such instances are used as keys in temporary BTree-based structures used only in a single program run. It can be disastrous if that BTree-based structure is stored to a database, though.

```
>>> class C:
...     pass
...
>>> a, b = C(), C()
>>> print a < b   # this may print 0 if you try it
1
>>> del a, b
>>> a, b = C(), C()
>>> print a < b   # and this may print 0 or 1
0
>>>
```

That example illustrates that comparison of instances of classes that don't define `__cmp__()` yields arbitrary results (but consistent results within a single program run).

Another problem occurs with instances of classes that do define `__cmp__()`, but define it incorrectly. It's possible but rare for a custom `__cmp__()` implementation to violate one of the three required formal properties directly. It's more common for it to "fall back" to address-based comparison by mistake. For example,

```
class Mine:
    def __cmp__(self, other):
        if other.__class__ is Mine:
            return cmp(self.data, other.data)
        else:
            return cmp(self.data, other)
```

It's quite possible there that the `else` clause allows a result to be computed based on memory address. The bug won't show up until a BTree-based structure uses objects of class `Mine` as keys, and also objects of other types as keys, and the structure is loaded from a database, and a sequence of comparisons happens to execute the `else` clause in a case where the relative order of object memory addresses happened to change.

This is as difficult to track down as it sounds, so best to stay far away from the possibility.

You'll stay out of trouble by follwing these rules, violating them only with great care:

1. Use objects of simple immutable types as keys in BTree-based data structures.
2. Within a single BTree-based data structure, use objects of a single type as keys. Don't use multiple key types in a single structure.
3. If you want to use class instances as keys, and there's any possibility that the structure may be stored in a database, it's crucial that the class define a `__cmp__()` method, and that the method is carefully implemented.

   Any part of a comparison implementation that relies (explicitly or implicitly) on an address-based comparison result will eventually cause serious failure.
4. Do not use `Persistent` objects as keys, or objects of a subclass of `Persistent`.

That last item may be surprising. It stems from details of how conflict resolution is implemented: the states passed to conflict resolution do not materialize persistent subobjects (if a persistent object P is a key in a BTree, then P is a subobject of the bucket containing P). Instead, if an object O references a persistent subobject P directly, and O is involved in a conflict, the states passed to conflict resolution contain an instance of an internal `PersistentReference` stub class everywhere O references P. Two `PersistentReference` instances compare equal if and only if they "represent" the same persistent object; when they're not equal, they compare by memory address, and, as explained before, memory-based comparison must never happen in a sane persistent BTree. Note that it doesn't help in this case if your `Persistent` subclass defines a sane `__cmp__()` method: conflict resolution doesn't know about your class, and so also doesn't know about its `__cmp__()` method. It only sees instances of the internal `PersistentReference` stub class.

Iteration and Mutation

As with a Python dictionary or list, you should not mutate a BTree-based data structure while iterating over it, except that it's fine to replace the value associated with an existing key while iterating. You won't create internal damage in the structure if you try to remove, or add new keys, while iterating, but the results are undefined and unpredictable. A weak attempt is made to raise `RuntimeError` if the size of a BTree-based structure changes while iterating, but it doesn't catch most such cases, and is also unreliable. Example:

```
>>> from BTrees.IIBTree import *
>>> s = IISet(range(10))
>>> list(s)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in s:  # the output is undefined
...     print i,
...     s.remove(i)
0 2 4 6 8
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
RuntimeError: the bucket being iterated changed size
>>> list(s)      # this output is also undefined
[1, 3, 5, 7, 9]
>>>
```

Also as with Python dictionaries and lists, the safe and predictable way to mutate a BTree-based structure while iterating over it is to iterate over a copy of the keys. Example:

```
>>> from BTrees.IIBTree import *
>>> s = IISet(range(10))
>>> for i in list(s.keys()):  # this is well defined
...     print i,
...     s.remove(i)
0 1 2 3 4 5 6 7 8 9
>>> list(s)
[]
>>>
```

BTree Diagnostic Tools

A BTree (or TreeSet) is a complex data structure, really a graph of variable-size nodes, connected in multiple ways via three distinct kinds of C pointers. There are some tools available to help check internal consistency of a BTree as a whole.

Most generally useful is the `BTrees.check` module. The `check.check()` function examines a BTree (or Bucket, Set, or TreeSet) for value-based consistency, such as that the keys are in strictly increasing order. See the function docstring for details. The `check.display()` function displays the internal structure of a BTree.

BTrees and TreeSets also have a `_check()` method. This verifies that the (possibly many) internal pointers in a BTree or TreeSet are mutually consistent, and raises `AssertionError` if they're not.

If a `check.check()` or `_check()` call fails, it may point to a bug in the implementation of BTrees or conflict resolution, or may point to database corruption.

Repairing a damaged BTree is usually best done by making a copy of it. For example, if *self.data* is bound to a corrupted IOBTree,

```
self.data = IOBTree(self.data)
```

usually suffices. If object identity needs to be preserved,

```
acopy = IOBTree(self.data)
self.data.clear()
self.data.update(acopy)
```

does the same, but leaves *self.data* bound to the same object.

A Resources

Introduction to the Zope Object Database, by Jim Fulton:
Goes into much greater detail, explaining advanced uses of the ZODB and how it's actually implemented. A definitive reference, and highly recommended.
http://www.python.org/workshops/2000-01/proceedings/papers/fulton/zodb3.html

Persistent Programing with ZODB, by Jeremy Hylton and Barry Warsaw:
Slides for a tutorial presented at the 10th Python conference. Covers much of the same ground as this guide, with more details in some areas and less in others.
http://www.zope.org/Members/bwarsaw/ipc10-slides

B GNU Free Documentation License

Version 1.1, March 2000

Preamble

The purpose of this License is to make a manual, textbook, or other written document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

B.1 Applicability and Definitions

This License applies to any manual or other work that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you".

A “Modified Version” of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A “Secondary Section” is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document’s overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (For example, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The “Invariant Sections” are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License.

The “Cover Texts” are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License.

A “Transparent” copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, whose contents can be viewed and edited directly and straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup has been designed to thwart or discourage subsequent modification by readers is not Transparent. A copy that is not “Transparent” is called “Opaque”.

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML designed for human modification. Opaque formats include PostScript, PDF, proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML produced by some word processors for output purposes only.

The “Title Page” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

B.2 Verbatim Copying

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

B.3 Copying in Quantity

If you publish printed copies of the Document numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a publicly-accessible computer-network location containing a complete Transparent copy of the Document, free of added material, which the general network-using public has access to download anonymously at no charge using public-standard network protocols. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

B.4 Modifications

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has less than five).
- State on the Title page the name of the publisher of the Modified Version, as the publisher.
- Preserve all the copyright notices of the Document.
- Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- Include an unaltered copy of this License.
- Preserve the section entitled "History", and its title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- In any section entitled "Acknowledgements" or "Dedications", preserve the section's title, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

- Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.
- Delete any section entitled “Endorsements”. Such a section may not be included in the Modified Version.
- Do not retitle any existing section as “Endorsements” or to conflict in title with any Invariant Section.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version’s license notice. These titles must be distinct from any other section titles.

You may add a section entitled “Endorsements”, provided it contains nothing but endorsements of your Modified Version by various parties – for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

B.5 Combining Documents

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections entitled “History” in the various original documents, forming one section entitled “History”; likewise combine any sections entitled “Acknowledgements”, and any sections entitled “Dedications”. You must delete all sections entitled “Endorsements.”

B.6 Collections of Documents

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

B.7 Aggregation With Independent Works

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, does not as a whole count as a Modified Version of the Document,

provided no compilation copyright is claimed for the compilation. Such a compilation is called an “aggregate”, and this License does not apply to the other self-contained works thus compiled with the Document, on account of their being thus compiled, if they are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one quarter of the entire aggregate, the Document’s Cover Texts may be placed on covers that surround only the Document within the aggregate. Otherwise they must appear on covers around the whole aggregate.

B.8 Translation

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License provided that you also include the original English version of this License. In case of a disagreement between the translation and the original English version of this License, the original English version will prevail.

B.9 Termination

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

B.10 Future Revisions of This Licence

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License ”or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.1 or any later version published by the Free Software Foundation; with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST. A copy of the license is included in the section entitled “GNU Free Documentation License”.

If you have no Invariant Sections, write “with no Invariant Sections” instead of saying which ones are invariant. If you have no Front-Cover Texts, write “no Front-Cover Texts” instead of “Front-Cover Texts being LIST”; likewise for Back-Cover Texts.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.