The following is from the 3rd Edition, and pertains to the Python 2.X version of ZODB. Its code has been changed to 3.X form, but was not tested due to the lack of a 3.X ZODB.

The ZODB Object-Oriented Database

ZODB, the Zope Object Database, is a full-featured and Python-specific object-oriented database system. ZODB can be thought of as a more powerful alternative to Python's shelves. It allows you to store nearly arbitrary Python objects persistently by key, like shelves, but it adds a set of additional features in exchange for a small amount of extra interface code.

ZODB is an open source, third-party add-on for Python. It was originally developed as the database mechanism for web sites developed with the Zope system mentioned in Chapter 12, but it is now available as a standalone package. It's useful outside the context of Zope as a general database management system in any domain.

Although ZODB does not support SQL queries, objects stored in ZODB can leverage the full power of the Python language. Moreover, in some applications, stored data is more naturally represented as a structured Python object. Table-based relational systems often must represent such data as parts scattered across multiple tables, associated with keys and joins.

Using a ZODB database is very similar to Python's standard library shelves, described in the prior section. Just like shelves, ZODB uses the Python pickling system to implement a persistent dictionary of persistent Python objects.

In fact, there is almost no database interface to be found—objects are made persistent simply by assigning them to keys of the root ZODB dictionary object, or embedding them in objects stored in the database root. And as in a shelve, records take the form of native Python objects, processed with normal Python syntax and tools.

Unlike shelves, ZODB adds features critical to some types of programs:

Concurrent updates

> You don't need to manually lock files to avoid data corruption if there are potentially many concurrent writers, the way you would for shelves.

Transaction commit and rollback

> If your program crashes, your changes are not retained unless you explicitly commit them to the database.

Automatic updates for some types of in-memory object changes

> Objects in ZODB derived from a persistence superclass are smart enough to know the database must be updated when an attribute is assigned.

Automatic caching of objects

> Objects are cached in memory for efficiency and are automatically removed from the cache when they haven't been used.

Platform-independent storage

> Because ZODB stores your database in a single flat file with large-file support, it is immune to the potential size constraints and DBM filesystem format differences of shelves. As we saw earlier in this chapter, a shelve created on Windows using BSD-DB may not be accessible to a script running with `gdbm` on Linux.

Because of such advantages, ZODB is probably worth your attention if you need to store Python objects in a database persistently, in a production environment. The only significant price you'll pay for using ZODB is a small amount of extra code:

- Accessing the database requires a small amount of extra boilerplate code to interface with ZODB—it's not a simple open call.
- Classes are derived from a persistence superclass if you want them to take advantage of automatic updates on changes—persistent classes are generally not completely independent of the database as in shelves, though they can be.

Considering the extra functionality ZODB provides beyond shelves, these trade-offs are usually more than justified for many applications.

A ZODB Tutorial

To sample the flavor of ZODB, let's work through a quick interactive tutorial. We'll illustrate common use here, but we won't cover the API exhaustively; as usual, search the Web for more details on ZODB.

Installing ZODB

The first thing we need to do is install ZODB on top of Python. ZODB is an open source package, but it is not a standard part of Python today; it must be fetched and installed separately. To find ZODB, either run a web search on its name or visit http://www.zope.org. Apart from Python itself, the ZODB package is the only component you must install to use ZODB databases.

ZODB is available in both source and self-installer forms. On Windows, ZODB is available as a self-installing executable, which installs itself in the standard *site-packages* subdirectory of the Python standard library (specifically, it installs itself in *C:\Python24\site-packages* on Windows under Python 2.4). Because that directory is automatically added to your module search path, no path configuration is needed to import ZODB's modules once they are installed.

Moreover, much like Python's standard pickle and shelve tools, basic ZODB does not require that a perpetually running server be started in order to access your database. Technically speaking, ZODB itself supports safe concurrent updates among multiple threads, as long as each thread maintains its own connection to the database. ZEO, an additional component that ships with ZODB, supports concurrent updates among multiple processes in a client/server context.

The ZEO distributed object server

More generally, ZEO, for Zope Enterprise Objects, adds a distributed object architecture to applications requiring high performance and scalability. To understand how, you have to understand the architecture of ZODB itself. ZODB works by routing object requests to a storage interface object, which in turn handles physical storage tasks. Commonly used storage interface objects allow for file, BerkeleyDB, and even relational database storage media. By delegating physical medium tasks to storage interface objects, ZODB is independent of the underlying storage medium.

Essentially, ZEO replaces the standard file-storage interface object used by clients with one that routes requests across a network to a ZEO storage server. The ZEO storage server acts as a frontend to physical storage, synchronizing data access among multiple clients and allowing for more flexible configurations. For instance, this indirection layer allows for distributing load across multiple machines, storage redundancy, and more. Although not every application requires ZEO, it provides advanced enterprise-level support when needed.

ZEO itself consists of a TCP/IP socket server and the new storage interface object used by clients. The ZEO server may run on the same or a remote machine. Upon receipt, the server passes requests on to a regular storage interface object of its own, such as simple local file storage. On changes, the ZEO server sends invalidation messages to all connected clients, to update their object caches. Furthermore, ZODB avoids file locking by issuing conflict errors to force retries. As one consequence, ZODB/ZEO-based databases may be more efficient for reads than updates (the common case for web-based applications).

Internally, the ZEO server is built with the Python standard library's `asyncore` module, which implements a socket event loop based on the `select` system call, much as we did in Chapter 12.

In the interest of space, we'll finesse further ZODB and ZEO details here; see other resources for more details on ZEO and ZODB's concurrent updates model. To most programs, ZODB is surprisingly easy to use; let's turn to some real code next.

Creating a ZODB database

Once you've installed ZODB, its interface takes the form of packages and modules to your code. Let's create a first database to see how this works:

```
...\PP4E\Dbase\ZODBscripts\> python
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage(r'C:\Mark\temp\mydb.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()
```

This is mostly standard code for connecting to a ZODB database: we import its tools, create a `FileStorage` and a `DB` from it, and then open the database and create the *root object*. The root object is the persistent dictionary in which objects are stored. `FileStorage` is an object that maps the database to a flat file. Other storage interface options, such as relational database-based storage, are also possible. When using the ZEO server configuration discussed earlier, programs import a `ClientStorage` interface object from the `ZEO` package instead, but the rest of the code is the same.

Now that we have a database, let's add a few objects to it. Almost any Python object will do, including tuples, lists, dictionaries, class instances, and nested combinations thereof. Simply assign your objects to a key in the database root object to make them persistent:

```
>>> object1 = (1, 'spam', 4, 'YOU')
>>> object2 = [[1, 2, 3], [4, 5, 6]]
>>> object2.append([7, 8, 9])
>>> object2
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>>
>>> object3 = {'name': ['Bob', 'Doe'],
               'age':  42,
               'job':  ('dev', 'mgr')}
>>>
>>> root['mystr']   = 'spam' * 3
>>> root['mytuple'] = object1
>>> root['mylist']  = object2
>>> root['mydict']  = object3
>>> root['mylist']
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

Because ZODB supports transaction rollbacks, you must commit your changes to the database to make them permanent. Ultimately, this transfers the pickled representation of your objects to the underlying file storage medium—here, three files that include the name of the file we gave when opening:

```
>>> import transaction
>>> transaction.commit()
>>> storage.close()

...\PP4E\Dbase\ZODBscripts> dir /B c:\mark\temp\mydb*
mydb.fs
mydb.fs.index
mydb.fs.tmp
```

Without the final commit in this session, none of the changes we made would be saved. This is what we want in general—if a program aborts in the middle of an update task, none of the partially complete work it has done is retained.

Fetching and changing

OK; we've made a few objects persistent in our ZODB. Pulling them back in another session or program is just as straightforward: reopen the database as before and index the root to fetch objects back into memory. The database root supports dictionary interfaces—it may be indexed, has dictionary methods and a length, and so on:

```
...\PP4E\Dbase\ZODBscripts\> python
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage(r'C:\Mark\temp\mydb.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()                         # connect
>>>
>>> print(len(root), root.keys())                    # size, index
4 ['mylist', 'mystr', 'mytuple', 'mydict']
>>>
>>> print(root['mylist'])                            # fetch objects
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
>>> print root['mydict']
{'job': ('dev', 'mgr'), 'age': 42, 'name': ['Bob', 'Doe']}

>>> root['mydict']['name'][-1]                       # Bob's last name
'Doe'
```

Because the database root looks just like a dictionary, we can process it with normal dictionary code—stepping through the keys list to scan record by record, for instance:

```
>>> for key in root.keys():
        Print(key.ljust(10), '=>', root[key])

mylist     => [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
mystr      => spamspamspam
mytuple    => (1, 'spam', 4, 'YOU')
mydict     => {'job': ('dev', 'mgr'), 'age': 42, 'name': ['Bob', 'Doe']}
```

Now, let's change a few of our stored persistent objects. When changing ZODB persistent class instances, in-memory attribute changes are automatically written back to the database. Other types of changes, such as in-place appends and key assignments, still require reassignment to the original key to force the change to be written to disk (built-in list and dictionary objects do not know that they are persistent):

```
>>> rec = root['mylist']
>>> rec.append([10, 11, 12])        # change in memory
>>> root['mylist'] = rec            # write back to db
>>>
>>> rec = root['mydict']
>>> rec['age'] += 1                 # change in memory
>>> rec['job'] = None
>>> root['mydict'] = rec            # write back to db

>>> import transaction
>>> transaction.commit()
>>> storage.close()
```

As usual, we commit our work before exiting Python or all our changes would be lost. One more interactive session serves to verify that we've updated the database objects; there is no need to commit this time because we aren't making any changes:

```
...\PP4E\Dbase\ZODBscripts\> python
>>> from ZODB import FileStorage, DB
>>> storage = FileStorage.FileStorage(r'C:\Mark\temp\mydb.fs')
>>> db = DB(storage)
>>> connection = db.open()
>>> root = connection.root()
>>>
```

```
>>> print(root['mylist'])
[[1, 2, 3], [4, 5, 6], [7, 8, 9], [10, 11, 12]]
>>>
>>> print(root['mydict'])
{'job': None, 'age': 43, 'name': ['Bob', 'Doe']}
>>>
>>> print(root['mydict']['age'])
43
```

We are essentially using Python as an interactive object database query language here; to make use of classes and scripts, let's move on to the next section.

Using Classes with ZODB

So far, we've been storing built-in object types such as lists and dictionaries in our ZODB databases. Such objects can handle rich information structures, especially when they are nested—a dictionary with nested lists and dictionaries, for example, can represent complex information. As for shelves, though, class instances open up more possibilities—they can also participate in inheritance hierarchies, and they naturally support record processing behavior in the form of class method functions.

Classes used with ZODB can either be standalone, as in shelves, or derived from the ZODB `Persistent` class. The latter scheme provides objects with a set of precoded utility, including the ability to automatically write instance attribute changes out to the database storage—no manual reassignment to root keys is required. To see how this works, let's get started by defining the class in Example 17-4: an object that records information about a bookmark in a hypothetical web site application.

Example ***Error! No text of specified style in document.****-1. PP4E\Dbase\ZODBscripts\zodb-class-make.py*

```
"""
define persistent class, store instances in a ZODB database;
import, call addobjects elsewhere: pickled class cannot be in __main__
"""

import time
mydbfile = 'data/class.fs'                          # where database is stored
from persistent import Persistent

class BookMark(Persistent):                         # inherit ZODB features
    def __init__(self, title, url):
        self.hits = 0
        self.updateBookmark(self, url)
    def updateBookmark(self, title, url):
        self.title = title                          # change attrs updates db
        self.url = url                              # no need to reassign to key
        self.modtime = time.asctime()

def connectdb(dbfile):
    from ZODB import FileStorage, DB
    storage = FileStorage.FileStorage(dbfile)       # automate connect protocol
    db = DB(storage)                                # caller must still commit
    connection = db.open()
    root = connection.root()
    return root, storage

def addobjects():
    root, storage = connectdb(mydbfile)
    root['ora']  = BookMark('Oreilly', 'http://www.oreilly.com')
    root['pp4e'] = BookMark('PP4E',    'http://www.rmi.net/~lutz/about-pp.html')
    import transaction
    transaction.commit()
    storage.close()
```

Notice how this class is no longer standalone—it inherits from a ZODB superclass. In fact, unlike shelve classes, it cannot be tested or used outside the context of a ZODB database. In exchange, updates to instance attributes are automatically written back to the database file. Also note how we've put connection logic in a function for reuse; this avoids repeating login code redundantly, but the caller is still responsible for keeping track of the root and storage objects and for committing changes on exit (we'll see how to hide these details better in the next section). To test, let's make a few database objects interactively:

```
...\PP4E\Dbase\ZODBscripts> python
>>> from zodb_class_make import addobjects
>>> addobjects()

...\PP4E\Dbase\ZODBscripts>dir /B data
class.fs
class.fs.index
class.fs.tmp
```

We don't generally want to run the creation code in the top level of our process because then those classes would always have to be in the module `__main__` (the name of the top-level file or the interactive prompt) each time the objects are fetched. Recall that this is a constraint of Python's pickling system discussed earlier, which underlies ZODB—classes must be reimported, and hence, located in a file in a directory on the module search path This might work if we load the class name into all our top-level scripts, with `from` statements, but it can be inconvenient in general. To avoid the issue, define your classes in an imported module file, and not in the main top-level script.

To test database updates, Example 17-5 reads back our two stored objects and changes them—any change that updates an instance attribute in memory is automatically written through to the database file.

*Example **Error! No text of specified style in document.**-2. PP4E\Dbase\ZODBscripts\zodb-class-read.py*

```
"""
read, update class instances in db; changing immutables like
lists and dictionaries in-place does not update the db automatically
"""

mydbfile = 'data/class.fs'
from zodb_class_make import connectdb
root, storage = connectdb(mydbfile)

# this updates db: attrs changed in method
print 'pp4e url:', root['pp4e'].url
print 'pp4e mod:', root['pp4e'].modtime
root['pp4e'].updateBookmark('PP4E', 'www.rmi.net/~lutz/about-pp4e.html')

# this updates too: attr changed here
ora = root['ora']
print 'ora hits:', ora.hits
ora.hits += 1

# commit changes made
import transaction
transaction.commit()
storage.close()
```

Run this script a few times to watch the objects in your database change: the URL and modification time of one is updated, and the hit counter is modified on the other:

```
...\PP4E\Dbase\ZODBscripts\> python zodb-class-read.py
pp4e url: http://www.rmi.net/~lutz/about-pp.html
pp4e mod: Mon Dec 05 09:11:44 2005
ora hits: 0

...\PP4E\Dbase\ZODBscripts> python zodb-class-read.py
pp4e url: www.rmi.net/~lutz/about-pp4e.html
pp4e mod: Mon Dec 05 09:12:12 2005
```

```
ora hits: 1

...\PP4E\Dbase\ZODBscripts> python zodb-class-read.py
pp4e url: www.rmi.net/~lutz/about-pp4e.html
pp4e mod: Mon Dec 05 09:12:24 2005
ora hits: 2
```

And because these are Python objects, we can always inspect, modify, and add records interactively (be sure to also import the class to make and add a new instance):

```
...\PP4E\Dbase\ZODBscripts> c:\python24\python
>>> from zodb_class_make import connectdb, mydbfile
>>> root, storage = connectdb(mydbfile)
>>> len(root)
2
>>> root.keys()
['pp4e', 'ora']
>>> root['ora'].hits
3
>>> root['pp4e'].url
'www.rmi.net/~lutz/about-pp4e.html'
>>> root['ora'].hits += 1
>>> import transaction
>>> transaction.commit()
>>> storage.close()

...\PP4E\Dbase\ZODBscripts> c:\python24\python
>>> from zodb_class_make import connectdb, mydbfile
>>> root, storage = connectdb(mydbfile)
>>> root['ora'].hits
4
```

A ZODB People Database

As a final ZODB example, let's do something a bit more realistic. If you read the sneak preview in Chapter 1, you'll recall that we used shelves there to record information about people. In this section, we bring that idea back to life, recoded to use ZODB instead.

By now, we've written the usual ZODB file storage database connection logic enough times to warrant packaging it as a reusable tool. We used a function to wrap it up in Example 17-4, but we can go a step further with object-oriented programming (OOP). As a first step, let's wrap this up for reuse as a component—the class in Example 17-6 handles the connection task, automatically logging in on construction and automatically committing changes on close. For convenience, it also embeds the database root object and delegates attribute fetches and index accesses back to the root.

The net effect is that this object behaves like an automatically opened and committed database root—it provides the same interface, but adds convenience code for common use cases. You can reuse this class for any file-based ZODB database you wish to process (just pass in your filename), and you have to change only this single copy of the connection logic if it ever has to be updated.

*Example **Error! No text of specified style in document.**-3. PP4E\Dbase\ZODBscripts\zodbtools.py*

```
class FileDB:
    "automate zodb connect and close protocols"
    def __init__(self, filename):
        from ZODB import FileStorage, DB
        self.storage = FileStorage.FileStorage(filename)
        db = DB(self.storage)
        connection = db.open()
        self.root = connection.root()
    def commit(self):
        import transaction
        transaction.commit()               # get_transaction() deprecated
    def close(self):
        self.commit()
        self.storage.close()
    def __getitem__(self, key):
        return self.root[key]              # map indexing to db root
    def __setitem__(self, key, val):
        self.root[key] = val               # map key assignment to root
    def __getattr__(self, attr):
        return getattr(self.root, attr)    # keys, items, values
```

Next, the class in Example 17-7 defines the objects we'll store in our database. They are pickled as usual, but they are written out to a ZODB database, not to a shelve file. Note how this class is no longer standalone, as in our earlier shelve examples—it inherits from the ZODB `Persistent` class, and thus will automatically notify ZODB of changes when its instance attributes are changed. Also notice the `__str__` operator overloading method here, to give a custom display format for our objects.

*Example **Error! No text of specified style in document.**-4. PP4E\Dbase\ZODBscripts\person.py*

```
"""
define persistent object classes; this must be in an imported
file on your path, not in __main__ per Python pickling rules
unless will only ever be used in module __main__ in the future;
attribute assignments, in class or otherwise, update database;
for mutable object changes, set object's _p_changed to true to
auto update, or manually reassign to database key after changes;
"""

from persistent import Persistent              # new module name in 3.3

class Person(Persistent):
    def __init__(self, name, job=None, rate=0):
        self.name = name
        self.job  = job
        self.rate = rate
    def changeRate(self, newrate):
        self.rate = newrate                    # auto updates database
    def calcPay(self, hours=40):
        return self.rate * hours
    def __str__(self):          myclass = self.__class__.__name__
        format = '<%s:\t name=%s, job=%s, rate=%d, pay=%d>'
        values = (myclass, self.name, self.job, self.rate, self.calcPay())
        return format % values

class Engineer(Person):
    def calcPay(self):
        return self.rate / 52   # yearly salary
```

Finally, Example 17-8 tests our `Person` class, by creating the database and updating objects. As usual for Python's pickling system, we store the class in an imported module, not in this main, top-level script file. Otherwise, it could be reimported by Python only when class instance objects are reloaded, if it is still a part of the module `__main__`).

*Example **Error! No text of specified style in document.**-5. PP4E\Dbase\ZODBscripts\person-test.py*

```
"""
test persistence classes in person.py; this runs as __main__, so the
classes cannot be defined in this file: class's module must be importable
when obj fetched; can also test from interactive prompt: also is __main__
"""

from zodbtools import FileDB                     # extended db root
from person import Person, Engineer              # application objects
filename = 'people.fs'                           # external storage
```

```
import sys
if len(sys.argv) == 1:                          # no args: create test records
    db = FileDB(filename)                       # db is root object
    db['bob'] = Person('bob', 'devel', 30)      # stores in db
    db['sue'] = Person('sue', 'music', 40)
    tom = Engineer('tom', 'devel', 60000)
    db['tom'] = tom
    db.close()                                  # close commits changes

else:                                           # arg: change tom, sue each run
    db = FileDB(filename)
    print db['bob'].name, db.keys()
    print db['sue']
    db['sue'].changeRate(db['sue'].rate + 10)    # updates db
    tom = db['tom']
    print tom
    tom.changeRate(tom.rate + 5000)             # updates db
    tom.name += '.spam'                         # updates db
    db.close()
```

When run with no command-line arguments, the test script initialized the database with two class instances: two `Person`s, and one `Engineer`. When run with any argument, it updates the existing database records, adding 10 to Sue's pay rate and modifying Tom's rate and name:

```
...\PP4E\Dbase\ZODBscripts> python person-test.py

...\PP4E\Dbase\ZODBscripts> python person-test.py -
bob ['bob', 'sue', 'tom']
<Person:         name=sue, job=music, rate=40, pay=1600>
<Engineer:       name=tom, job=devel, rate=60000, pay=1153>

...\PP4E\Dbase\ZODBscripts> python person-test.py -
bob ['bob', 'sue', 'tom']
<Person:         name=sue, job=music, rate=50, pay=2000>
<Engineer:       name=tom.spam, job=devel, rate=65000, pay=1250>

...PP4E\Dbase\ZODBscripts> python person-test.py -
bob ['bob', 'sue', 'tom']
<Person:         name=sue, job=music, rate=60, pay=2400>
<Engineer:       name=tom.spam.spam, job=devel, rate=70000, pay=1346>
```

Notice how the `changeRate` method updates Sue—there is no need to reassign the updated record back to the original key as we have to do for shelves, because ZODB `Persistent` class instances are smart enough to write attribute changes to the database automatically on commits. Internally, ZODB's persistent superclasses use normal Python operator overloading to intercept attribute changes and mark the object as changed.

However, direct in-place changes to mutable objects (e.g., appending to a built-in list) are not noticed by ZODB and require setting the object's `_p_changed`, or manual reassignment to the original key, to write changes through. ZODB also provides custom versions of some built-in mutable object types (e.g., `PersistentMapping`), which write changes through automatically.

ZODB Resources

There are additional ZODB concepts and components that we have not covered and do not have space to discuss in detail in this book. For instance, because ZODB stores objects with Python's `pickle` module, all of that module's constraints discussed earlier in this chapter apply. Moreover, we have not touched on administrative requirements. Because the `FileStorage` interface works by appending changes to the file, for example, it requires periodically running a utility to pack the database by removing old object revisions.

For more about ZODB, search for ZODB and Zope resources on the Web. Here, let’s move on to see how Python programs can make use of a very different sort of database interface—relational databases and SQL.