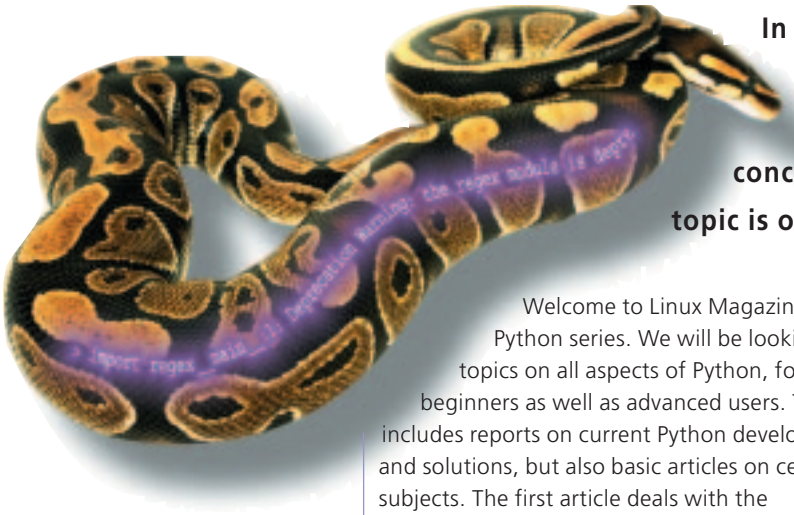
Object persistence in Python

PYTHON POWER

ANDREAS JUNG



In a new series on Python, Linux Magazine will be reporting on current developments every other month and introducing the concepts that make Python unique. Our first topic is object persistence.

Welcome to Linux Magazine's new Python series. We will be looking at topics on all aspects of Python, for beginners as well as advanced users. This includes reports on current Python developments and solutions, but also basic articles on certain subjects. The first article deals with the permanent storage of objects. But first a brief overview of Python...

Python is a scripting language that was developed at the beginning of the 90s by Guido van Rossum and has since evolved into a universally employed programming language. Today, Python is the most important and most widely used scripting language apart from Perl. As this description implies, Python is an interpreted language, compilation of Python programs takes place at runtime.

A magazine article cannot hope to give a full introduction to Python, however, we will discuss some of Python's concepts and advantages. A detailed introduction can be found in the Python

The most important innovations in Python 2.1

Nested scopes

Until Python 2.0 there were three name spaces, which are searched for variables in the following order: local name space, module name space and built-in name space. This separation is not intuitive if you look at nested functions:

```
def f():
   ...
   def g(value):
      ...
      return g(value-1)
```

Invocation of function *g()* in the return statement will cause a name error exception, because *g* has not been defined in any of the three name spaces. Python 2.1 removes this shortcoming and allows the nesting of name spaces through importing the new *nested_scopes* module.

__future__ statements

New features are introduced with every version of Python. This may lead to a break in compatibility with existing applications. In order to alleviate this problem, new aspects that will become standard features in Python 2.2 can be linked using a *__future__* import statement. Nested scopes will become a standard feature of Python from version 2.2 onwards. Although their implementation is already finished they have not yet been enabled in Python 2.1. To be able to use them anyway, they need to be linked and enabled with

```
from __future__ import nested_scopes
```

Warning framework

Over the years many modules have accumulated that are no longer supported, are obsolete or have been replaced by newer ones with improved functionality. It is difficult for developers to remove modules

tutorial at *http://www.python.org/doc*, but also in the new Open Source book Dive Into Python (*http://www.diveintopython.org*).

Python overview

We'd briefly like to mention a few of Python's advantages and distinctive features:

Quick to learn and to easy read: Python's syntax is simple and easy to understand, and its functionality is clear. Unlike Perl, even very large projects can be created, maintained and still be intelligible some time later. Its language range is orthogonal, as a rule there is no duplication of functionality. Loops are realised as *for* or *while constructs, repeat* or *do..until* loops are unnecessary.

Modular: related functionalities (for example sockets or graphical options) have been combined into modules and are imported when required. In the spirit of code reuse, modules can be used by different applications.

Interactive: Python has an interactive mode, which makes familiarisation very easy, particularly for beginners.

Compact: Compared to compiled languages like C or C++ Python programs are very compact. Python's data types, such as dictionaries, lists and tuples, allow most complex operations to fit onto one line. Programs are structured into codeblocks by indentation of the source code. The bracketing familiar from C is therefore redundant.

Object-oriented: In contrast to other programming languages Python was designed from the outset to be object-oriented rather than being extended with object-oriented concepts later on, like Perl, for example. This unified concept distinguishes Python significantly from its competitors.

Python's increasing popularity has one main reason: the clear and simple language structure makes it easily accessible. In the meantime, Python is used by schools and universities to teach programming skills.

However, Python is not only of interest to beginners, it is the Swiss Army knife of programming. It is used in areas as diverse as Web applications, string processing, administrative and other applications, numeric calculations and controlling complex production environments in factories. Python inherently offers many useful concepts that are not found in other languages, for example object persistence.

What is object persistence?

In every object-oriented programming language objects contain methods and attributes. For many applications it is desirable to deposit an object permanently on a storage medium in order to reuse it after restarting the program. Following a program termination the latest stored state of the object can then be accessed.

It is always possible to write application-dependant code for the export of important data, but each modification of the object also requires the export functionality to be amended accordingly. What is required at this point is transparent object persistence, that is, a mechanism that allows objects to be stored permanently without additional code. This should happen without necessitating the programmer or the application to have special knowledge about persistence.

Python persistence

For a long time Python has contained *pickle* and *cPickle*, with which objects can be serialised. Objects are serialised into character streams, which

Tuples: A tuple is a number of values separated by commas

without running the risk that applications won't work with later versions.

The warning framework makes it possible to issue version-dependant warnings that a module will no longer be contained in the next version or a functionality will be changed or removed. For example, when importing the regex module, Python 2.1 issues the warning:

```
> import regex
__main__:1: Deprecation Warning: the regex module is
deprecated; please use
the re module
```

Users then have one release cycle to convert their software to the newer module re for regular expressions.

Function attributes

In Python 2.1 attributes can be assigned to functions:

```
def func():
    ....
func.author = "Holger Müller"
func.security = 1
```

All attributes are stored in the function's dictionary *__dict__*. Until version 2.0 it was only possible to hide additional information in the doc string, which could be read through *f.__doc__*.

New installation mechanism

From version 2.1 the installation is carried out using the distutils package, which is the standard installation tool for Python modules.

It is therefore no longer necessary to go to the trouble of configuring the modules manually, as was the case in older versions. The installation script checks automatically which modules it can compile (similar to *configure), based on the headers and libraries, and then builds them automatically.*

Listing 1: Pickling an object

```
#Import the pickle module
import cPickle
class myClass:
       def_init_(self,num,txt):
                self.num=num
                self.txt=txt
#Generating a myClass object
instClass=myClass(212,'Python is cool')
fname='instClass.p'
#Serialising instClass into a file
cPickle.dump(instClass,open(fname,'w'))
#Open file with pickled object and
#create a new object
newinstClass=cPickle.load(open(fname,'r'))
print newinstClass.num,newinstClass.txt
```

Listing 2: Opening a ZODB database

```
from ZODB import DB, FileStorage
fstorage = FileStorage.FileStorage(`Data.fs')
db = DB(fstorage)
connection = db.open()
root = connection.root()
```

can then be in files. This process is called pickling; an object is conserved, as it were, in order to be reused later. Alternatively, Python can read in such a serialized object and convert it back into an object (unpickling).

Both modules are identical in their functionality: *cPickle* is the C reimplementation of the *pickle* module written in Python, and is always preferable for efficiency reasons.

In the example in Listing 1 an object with the two attributes *num=212* and *txt='Python is cool'* is created. The object is stored permanently in an internal format in *instClass.p* by invoking *cPickle.dump()*. The subsequent call *cPickle.load()* loads the files and generates a new instance of *myClass*, which has the same attributes as the original object.

This approach is generally possible for every Python object, however, there are some exceptions. For example, file objects or sockets cannot be serialized, which would not be sensible anyway. Pickling allows persistent storage of any object – even ones with multiple inheritance – but the programmer still has to implement parts of the code himself.

Persistence in Python using the ZODB

Based on the pickle mechanism, the Zope Object Data Base (ZODB for short) was created during the development of the Zope application server. It frees the developer from the burden of implementation as well. Its use is relatively simple: to the developer the ZODB appears as a mapping object which is addressed in the same way as a Python dictionary:

```
zodb[`instClass'] = instClass
```

The object is bound to the key *`instClass'* in the ZODB and stored. In the same way objects can easily be retrieved from the ZODB:

```
instClass = zodb[`instClass']
```

That looks very elegant, and it is. But before we get to that point, ZODB has to be installed first. ZODB is not restricted to a specific medium for storing objects. Normally it deposits objects in a file within the file system, however, adapters for databases such as Oracle or BerkleyDB exist. The storage medium is transparent to the application. Only when opening the ZODB does the medium have to be specified, that is, if applicable, the underlying actual database layer.

Installation of the ZODB

The ZODB is integrated into Zope and can be used if Zope has already been installed. If you don't need Zope, there is a stand-alone version of the ZODB which is being maintained by A M Kuchling.

After unpacking the archive the installation is performed using the distutils tool (contained in Python 2.0/2.1, for Python 1.5.x the distutils have to be installed separately):

```
python setup.py install
```

That should automatically compile and install all ZODB sources and modules. It is advisable to use the current version of Python, 2.1.

Using the ZODB

How to open the ZODB when using a file as the storage medium can be seen in detail in Listing 2. The *FileStorage* object in this case represents the storage medium that is being used for the ZODB. When using a ZODB adapter for a relational database the call must be amended accordingly. The subsequent calls open the database and create the actual *`root'* object through which the ZODB is addressed by the application.

Serialisable Python objects can now easily be deposited in the ZODB:

```
root[`red'] = `ZODB is cool'
root[`blue'] = [`Perl','is','cool']
```

Assignment only stores the objects in the ZODB temporarily. In order to store them persistently – that is permanently – the transaction must be committed:

```
get_transaction().commit()
```

A transaction is an atomic operation and consists of

Listing 3: creating persistent classes

```
import ZODB
import Persistence
class PLanguage(Persistence.Persistent)
   def __init__(self,lang,easy2learn):
      self.language   = lang
      self.learneffort = easy2learn
      self.authors    = []
....
languages = []
languages.append( PLanguage(`Python',`very easy') )
languages.append( PLanguage(`Perl',`very hard') )
languages.append( PLanguage(`TCL',`easy') )
zodb[`languages'] = languages
TCL = zodb[`languages][2]
TCL.learneffort = `not easy'
get_transaction().commit()
```

a sequence of changes within the database. The transaction mechanism of the database ensures that either all changes are carried out or none. This guarantees data integrity between two commit calls.

After the data have been stored in the ZODB they can, of course, be retrieved. Opening the ZODB is done in the same way as the writing of data.

Reading the data is identical to using a dictionary:

```
print root[`red']   -> `ZODB is cool'
print root[`blue']   -> [`Perl',`is',`cool']
```

Changes in the ZODB

The ZODB automatically recognises changes to objects and also stores them, with one exception: changes to lists and dictionaries are not recognised automatically. That is true on a general level for all objects that are described as mutable, or changeable, in the Python philosophy.

Changes to a list or a dictionary must therefore not be made using

```
root[`blue'].append(`a lot')
     get_transaction().commit()
```

but instead require a new assignment of the object:

```
temp = root[`blue']
     temp.insert(2,'not')
     root[`blue'] = temp
     get_transaction().commit()
```

Persistent classes

Converting classes into persistent classes is particularly easy. They simply need to be derived from the class *Persistence.Persistent*. The process in detail is illustrated by the example in Listing 3.

As explained above, changes to mutable data types are not automatically recognised by the ZODB. In such cases alterations have to be explicitly indicated to the database by setting the attribute *_p_changed* to *1*. The ZODB will then update the object accordingly:

```
class PLanguage(Persistence.Persistent)
  ....
   def setAuthor(self,author):
      self.authors.append( author )
      self._p_changed = 1
```

Outlook

The Zope extension Zope Enterprise Objects (ZEO) can be used to build a distributed ZODB, which means objects can also be stored distributed.

This article shows how easy the ZODB is to use and that it represents a powerful tool for Python developers, which allows transparent object persistence while requiring little effort to learn and only minor source code amendments.

The author

Andreas Jung lives near Washington D.C. and works for Zope Corporation (formerly Digital Creations) as a software engineer in the Zope core team. Email: andreas@andreas-jung.com

Info

Python in practice:	*http://www.python.org/psa/Users.html*
ZODB pages by A. M. Kuchling:	*http://www.amk.ca/zodb/*
M. Pelletier: ZODB for Python Programmers:	*http://www.zope.org/Documentation/Articles/ZODB1*
Zope Enterprise Objects (ZEO):	*http://www.zope.org/Products/ZEO*