IST FP6-004779

PYPY

Researching a Highly Flexible and Modular Language Platform and Implementing it by Leveraging the Open Source Python Language and Community

STREP

IST Priority 2

D03.1 Synchronization with Standard Python and the Extension Compiler

Due date of deliverable: March 2007

Actual Submission date: March 23th, 2007

Start date of Project: 1st December 2004 **Duration: 28 months**

Lead Contractor of this WP: HHU

Authors: Armin Rigo (HHU)

Revision: Final

Project co-funded by the European Commission within the Sixth Framework Programme (2002-2006)

Dissemination Level: PU (Public)

PyPy D03.1: Extension Compiler

Revision History

Date	Name	Reason of Change
2006-06-16	Armin Rigo, Gerald Klix	Initial Extension Compiler documentation on-line
2006-12-12	Armin Rigo	Report on task 1
2007-01-19	Armin Rigo	Work on the 2nd RCtypes implementation
2007-01-22	Armin Rigo	Publishing of intermediate version on the web
2007-02-27	Guido Wesdorp	Some fixes of grammar and typos
2007-02-28	Carl Friedrich Bolz	Publish Interim on the Web Page
2007-03-21	Armin Rigo	Reviewer comments integrated
2007-03-21	Carl Friedrich Bolz	Publish Final version on the Web Page

Abstract

This report describes the work and results achieved in WP03: Synchronization with Standard Python. In particular, we present the RPython Extension Compiler, which allows programmers to write extension modules for multiple Python implementations as a single RPython source. So far, PyPy's translation toolchain can compile such modules either together with the rest of the PyPy interpreter (in which the module ends up as a new built-in module) or as C code that conforms to the C API defined by CPython (producing in that case a dynamically-loaded extension in the CPython style).

Purpose, Scope and Related Documents

This document describes:

- The work done to follow the core CPython language evolution.
- The tools that we developed in order to enable third-party extension module writers to port existing or new modules to PyPy. This report also includes the documentation that we developed in order to guide people in this task: this can be found in Appendix 2.

This document describes the procedures we use to track the CPython language evolution, and lists the main new language features that we needed to adapt PyPy to during the course of the funded project.

It explains and describes the approach we selected to support and attract third-party extension modules. It points to (but is not meant to act as) general user documentation for third-parties. It gives some implementation details on the main parts of the Extension Compiler as well as RCtypes, the foreign function call implementation.

Part two of this document assumes basic knowledge of the PyPy architecture [D04.2] and translation process [D05.1]. The following on-line documentation may give more up-to-date information on these topics:

- http://codespeak.net/pypy/dist/pypy/doc/architecture.html
- http://codespeak.net/pypy/dist/pypy/doc/translation.html

Contents

Appendices

1. What's New in Python 2.4 from the CPython community [PY24]
2. Extension compiler documentation from the web page

1 Executive Summary

In the field of Python implementations, PyPy is a direct "competitor" to CPython, the existing well-established Python interpreter written in C. One of the main conditions for PyPy to gain acceptance in the community of Python programmers is to keep close to CPython in term of supported features. These features can be naturally divided in two categories: core language features - which are updated at a relatively slow pace by the designers of Python and implemented by them in CPython - and extension modules. The latter are third-party extensions to the CPython interpreter which add new functionality, visible to the programmer as extra modules performing tasks that are not possible or easy to achieve in pure Python code. (A third category of essential feature for Python programmers is the large amount of readily available pure Python libraries, but these should work out-of-the-box on any compliant Python implementation.)

The present report thus covers the tasks of keeping in sync with CPython on the following two fronts:

- Core language features:

  Task 1 of WP03 set out to follow the core CPython language evolution. The Python language went from 2.3 to 2.4 to 2.5 in the past two years. We chose CPython 2.4.1 as our target language. We incorporated a few syntactic features from 2.5, although this is a very recent release (September 2006) and not yet widely relied upon.

- Extension modules:

  PyPy's tool-chain is able to translate a subset of the Python language, RPython [D05.1], to other (typically lower-level) languages and platforms. PyPy's own extension modules are written in RPython and based on the Object Space API, part of the core of the PyPy interpreter's approach [D04.2]. The long-term approach that we selected and implemented to ease synchronization of the multiple Python implementations is to make it convenient for developers of future CPython modules to write such modules in RPython as well. For this purpose, we offer a stand-alone tool, the Extension Compiler, based on our translation toolchain and able to compile RPython Extension Modules to CPython Extension Modules. The same RPython Extension Modules can be compiled within PyPy.

2 Core language features

2.1 Python releases

During the past two years, the following CPython versions were released:

- Python 2.4 (November 2004)
- Python 2.3.5 (February 2005)
- Python 2.4.1 (March 2005)
- Python 2.4.2 (September 2005)
- Python 2.4.3 (March 2006)
- Python 2.5 (September 2006)
- Python 2.4.4 (October 2006)
- Python 2.3.6 (October 2006)

Last-digit-revisions correspond to bug fix releases. It is customary for a Python version to only become widely accepted and relied upon after the ".1" or ".2" release has been made. When PyPy was started, we followed the most recent version at the time, which was the 2.3.x branch.

Predicting the above timeline it was clear from the beginning that we would have to switch to the 2.4 branch. The corresponding work was achieved during summer 2005, when we decided that PyPy should target for the foreseeable future compatibility with Python 2.4.x, and more precisely CPython 2.4.1.

The highlights of this work precisely correspond to the document *What's New in Python 2.4* [PY24], included in Appendix 1.

In prevision of the Python 2.5 release we incorporated a few of the syntactic extensions that the Python 2.5 language defines, namely Conditional Expressions [PEP308] and the `with` statement [PEP343]. Some other 2.5 language changes are clean-ups of historical accidents of the Python development and have always been available in PyPy, e.g. Exceptions as New-Style Classes [PEP352] and container sizes not limited to 32-bit numbers on 64-bit machines ("Using ssize_t as the index type" [PEP353]). Finally, some features are yet to be implemented, e.g. the new `__index__` method [PEP357].

It should be noted that at least for the new features that we implemented so far, the expressiveness of Python compared to C and the flexibility of relevant parts of the design (e.g. our bytecode compiler) made the task fairly easy.

Nevertheless, we decided to not switch our primary target from Python 2.4.1 to Python 2.5 yet for a number of reasons, in addition to the practical reason that 2.5 was only released late during PyPy's project (near the start of our last phase). The main reason is that Python 2.5 is not yet widely deployed, so that programmers still restrain from using the new features for the time being. As occurred with 2.4, it is likely that PyPy will switch to the 2.5.x branch when Python reaches a more mature stage, probably after the release of version 2.5.1 or 2.5.2.

In summary, the main technical work that classifies under WP03 Task 1 has been the switch from targeting version 2.3 to version 2.4 of the language, which occurred soon after the start of the project's funding. More recently, we integrated some features from version 2.5 but on a minor scale, choosing instead to keep version 2.4 as our primary target for the time being.

2.2 CPython community collaboration

A high proportion of the PyPy developers turned out to be regular CPython contributors and community members. Collaboration between PyPy and the rest of the community was successfully carried on by these persons, both on a personal level and via mailing list interactions. To reach the larger community, we presented multiple aspects of PyPy in one or several talks at various conferences, including the major Python conferences. Technical talks have been given at the following events:

- 21C3 (21st Chaos Communication Congress), 2004
- 22C3 (22nd Chaos Communication Congress), 2005
- ACCU 2005
- ACCU 2006
- DLS (Dynamic Languages Symposium at OOPSLA), 2006
- EuroPython 2004
- EuroPython 2005
- EuroPython 2006
- LaTe Lunch, Cardiff, 2006

- OSCON (O'Reilly Open Source Convention), 2003
- PyCon (Python Conference) 2004
- PyCon (Python Conference) 2005
- PyCon (Python Conference) 2006
- Vancouver Python Workshop, 2006

Note that 2003 and 2004 conferences are not part of the present EU project; we mention them because the talks were given by people now part of the EU project. Not listed above, though, are talks from people outside the EU project that talked about PyPy at conferences that we could not attend, e.g. OSDC 2006 (Australia). Most of these talks have triggered interesting collaborations between EU project members and non-EU project members about PyPy.

On another level: for the purpose of tracking the changes occurring in CPython, any relevant change - both at the language specification level and at the level of "semi-internal" details that users might nevertheless rely upon - is unlikely to get simply overlooked. Such changes are either well-documented or well known to the members of the PyPy project since many are also active CPython contributors. For this reason we judged it superfluous to develop ways to automate the process of tracking CPython developments beyond the usual tools: CPython's mailing list, bug and patch trackers, and revision control system.

3 Extension modules

In regular Python, the ability for users to write external modules is of great importance. So far, these external modules must be written in C, which is both an advantage - it allows access to external C libraries, low-level features, or raw performance - and an inconvenience.

For CPython, two mainstream solutions already exist to help alleviate the burden of writing C code by hand: Pyrex [PYREX], which translates modules written in a static optionally-typed Python-like language into C code suitable for CPython (the language is not a subset of Python, though); and a family of tools like SWIG [SWIG] which helps connect C and C++ libraries with Python and other high-level languages. SWIG turns a single C- or C++-level interface definition into boilerplate wrapper code for each of these high-level languages.

In the context of PyPy the most immediate drawback of the C language for external modules is that low-level details like multithreading (e.g. locks) and memory management must be written explicitly. This would prevent the same module from being used in several differently compiled versions of `pypy-c`. The most natural solution is to write PyPy's extension modules in the same high-level language as the rest of the interpreter, i.e. RPython [D05.1]. In this way, the modules naturally integrate with our translation toolchain and can be turned into the correctly-flavoured low-level code along with the rest of `pypy-c`.

As an extension of this idea, we can also produce low-level implementations of the same modules for other Python implementations, just by changing specific aspects of the translation process. In order to make this approach an attractive alternative to using C, as well a new tool in the toolbox of third-party extension module writers, we must thus:

- provide a tool, the Extension Compiler, based on the translation toolchain and targeting various Python implementation;
- ensure that RPython is usable for extension modules: it must allow access to external C libraries and low-level features (it already provides reasonable performance). This is the goal of RCtypes.

3.1 The Extension Compiler

Given that RPython is high-level and flexible in what platforms and languages it can be compiled to, we have made it possible for PyPy extension modules written in RPython to be compiled to C code that defines a regular CPython extension module, following the CPython API. This is accomplished by the Extension Compiler. The same approach could be adapted to target other Python implementations as well (Jython, IronPython), enabling a one-source-fits-all approach.

3.1.1 Mixed modules

PyPy defines a format for extension module writers: the so-called *mixed module* approach. A single mixed module is implemented as a set of Python source files, forming a subpackage of the *pypy/module/* package. Such a subpackage is the definition of a single module, which has its interface specified in the `__init__.py` of the subpackage, and of which the implementation is loaded as needed from the various other `.py` files of the subpackage. The name "mixed" comes from the fact that they mix interpreter-level and application-level submodules to present a single, coherent module to the user. This format was introduced and documented in [D04.2] already.

The recent work in WP03 has been to build tools allowing mixed modules written in this format to be reused for new purposes, including being compiled to a CPython C extension module. More precisely, each mixed module can be used in four different ways:

- It can be run directly on top of CPython. This uses a specific Object Space, called the CPy Object Space, which directly runs all space operations. A module run in this way does not use the rest of the PyPy machinery (Standard Interpreter and Standard Object Space). This mode is useful for testing.
- It can be compiled as a CPython extension module. The *Extension Compiler* is the stand-alone tool that can be invoked in a single shell command, producing a dynamically-loadable module for CPython.
- It can be run with PyPy on top of CPython. This mode gives a complete PyPy environment in which the mixed module can be tested.
- It can be compiled together with PyPy. It then becomes a built-in module of the generated `pypy-c`.

3.1.2 User documentation

More details about how to implement, test and translate a mixed module for both PyPy and CPython are given in our on-line documentation and are beyond the scope of the present report. The main guide, put forward in the documentation index page, is available at http://codespeak.net/pypy/dist/pypy/doc/extcompiler.html and also included in Appendix 2 of the present report.

The guide refers to two other documents in particular that are essential reading for third-party extension module writers: the coding guide describing the RPython language, and our RCtypes documentation.

3.1.3 Implementation overview

A typical example of a mixed module is a wrapper for a C library. In this case, the mixed module contains external function calls to access the C functions, and possibly C variables and data structures. (The details of how this is done in RPython are documented in the RCtypes section below.)

When using the Extension Compiler, there is an additional source of external function calls: it is the *CPy Object Space*, the special Object Space on which the Extension Compiler is based. Recall from [D04.2] that interpreter-level code manipulates application-level objects via an API on a object called the Object Space: for example, the

`space.add()` method is responsible for performing the details of the addition of two arbitrary user objects. All the interpreter-level parts of mixed modules are written in that style: they manipulate application-level objects exclusively through space operations. The Extension Compiler can now compile the mixed module in a mode in which the provided space is a CPy Object Space instead of the regular Standard Object Space. The property of the CPy Object Space is that its API implementation is essentially just external function calls to the C functions of the CPython API.

This means that the RPython sources of both the mixed module and the CPy Object Space contain dynamic foreign function calls to external C code. However, when PyPy's translation toolchain turns this RPython source into C code, RCtypes transforms these dynamic calls into static calls - the generated C code only contains static, regular C calls to the external C functions.

3.1.4 Related work

Our approach is related to the foreign language interfaces of various existing language implementations. For example, GNU CLISP [CLISP] provides a Foreign Function Call Facility [DFFI] in which C functions and data structures are declared in a regular Lisp file. The Lisp file can be either loaded as usual - in that case the calls are done via a dynamic foreign function interface, similar to ctypes [CTYPES] on which RCtypes is based - or turned into a C file that can be compiled and linked with the Lisp system - resulting in static, regular C function calls in the C files.

An original aspect of our approach is that we started from a dynamic foreign function call interface (ctypes) that was never designed with static compilation in mind, and integrated its translation to C with the rest of our RPython translation toolchain. Also, as far as we know, no other project is using the same foreign function call interface to generate not only the static calls to the external libraries, but also the static calls to the virtual machine's own gluing API (as the CPy Object Space does with the CPython API). More experience is needed before we can see the full pros and cons of this approach.

On a more general level, our approach can be compared to Pyrex in that we translate a Python-like language to low-level code; however, unlike Pyrex, RPython is actually a subset of Python and requires no type annotations in the source to run efficiently. Even the `ctypes` module allowing access to external C libraries is an official part of Python since CPython 2.5. This means that the RPython modules can be directly tested, debugged and even used on top of an out-of-the-box Python interpreter. The translation step is optional: it is a way to recover performance on top of - and integrate more closely with - multiple Python implementations.

3.2 RCtypes

ctypes [CTYPES] is a library for CPython that has been around for some years, and that is now integrated in Python 2.5. It allows Python programs to directly perform calls to external libraries written in C, by declaring function prototypes and all necessary data structures dynamically in Python. It is based on the platform-dependent *libffi* library [LIBFFI].

RCtypes is ctypes restricted to be used from RPython programs. Basically, an RPython program that is going to be translated to C or a C-like language can use (a subset of) ctypes in a more or less normal fashion.

RCtypes is particularly useful in conjunction with mixed modules and the Extension Compiler: it lets the RPython part of mixed modules use C resources directly, just like ctypes lets normal Python programs use C resources. The difference, however, is that during the translation of the RPython module or program to C, the source code - which contains dynamic foreign function calls written using ctypes - becomes static C code that performs the same calls. In other words, once compiled within an RPython program, ctypes constructs are all replaced by regular, static C code that no longer uses the *libffi* library. This makes ctypes a very good way to interface with any C library -- not only from plain Python, but from any program or extension module that is written in RPython.

This document assumes some basic familiarity with ctypes. It is recommended to first look at the ctypes documentation at [CTYPES]. More details about both ctypes and RCtypes are also available in our on-line documentation at http://codespeak.net/pypy/dist/pypy/doc/rctypes.html.

3.2.1 Restrictions

The main restriction that RCtypes adds over ctypes is similar to the general restriction that RPython adds over Python: you cannot be "too dynamic" at run-time, but you can be as dynamic as you want during bootstrapping, i.e. when the source code is imported.

For RCtypes, this means that all ctypes type and function declarations must be done while bootstrapping. You cannot declare new types and functions from inside RPython functions that are going to be translated to C. In practice, this is a fairly soft limitation: most modules using ctypes would typically declare all functions globally at module-level anyway.

Apart from this, many ctypes constructs can be freely used: pointers, structures, arrays, external functions and all primitive types except for c_float, c_wchar and c_wchar_p (they will be added at a later time). The following ctypes functions are supported at run-time: create_string_buffer(), pointer(), POINTER(), sizeof() and cast(). (Adding support for more is easy.) There is some support for Unions, and for callbacks and a few more obscure ctypes features, but e.g. custom allocators and return-value-error-checkers are not supported.

3.3 RCtypes implementation design

We have experimented with two different implementation approaches. The first one is relatively mature, but does not do the right thing with memory management issues in the most advanced cases. Moreover, it prevents the RPython program to be compiled with a Moving Garbage Collector, which has been the major factor stopping us from experimenting with advanced GCs so far. It is described in the section RCtypes implemented in the RTyper.

The alternative implementation of RCtypes is work in progress; its approach is described in the section RCtypes implemented via a pure RPython interface.

3.3.1 RCtypes implemented in the RTyper

The currently available implementation of RCtypes works by integrating itself within the annotation and most importantly RTyping process of the translation toolchain [D05.1]: the ctypes objects that the RPython program uses, and the operations it performs on them, are individually replaced during RTyping by sequences of low-level operations performing the equivalent operation in a C-like language.

3.3.2 Annotation

Ctypes on CPython tracks all the memory it has allocated by itself, may it be referenced by pointers and structures or only pointers. Thus memory allocated by ctypes is properly garbage collected and no dangling pointers should arise.

Pointers to structures returned by an external function are a different story. For such pointers we have to assume that they were not allocated by ctypes, and just keep a pointer to the memory, hoping that it will not go away. This is the same in RCtypes.

For objects that are known to be allocated by RCtypes, a simple GC-aware box suffices to hold the needed memory. For other objects, an extra indirection is required. Thus we use the annotator to track which objects are statically known to be allocated by RCtypes.

3.3.3 RTyping: Memory layout

In ctypes, all instances are mutable boxes containing either some raw memory with a layout compatible to that of the equivalent C type, or a reference to such memory. The reference indirection is transparent to the user; for example, dereferencing a ctypes object "pointer to structure" results in a "structure" object that doesn't include a copy of the data, but only a reference to that data. (This is similar to the C++ notion of a reference: it is just a pointer at the machine level, but at the language level it behaves like the object that it points to, not like a pointer.)

The rtyper maps this to the LLType model as follows. For boxes that embed the raw memory content (i.e. are known to own their memory):

```
Ptr( GcStruct( "name_owner",
       ("c_data", Struct(...) ) ) )
```

where the raw memory content and layout is specified by the "Struct(...)" part.

The key point here is that the "Struct(...)" part follows exactly the C layout, with no extra headers. If it contains pointers, they point to exact C data as well, without headers, and so on. From C's point of view this is all perfectly legal C data in the format expected by the external C library -- because the C library never sees pointers to the whole GcStruct boxes.

In other words, we have a two-level picture of what data looks like in a translated RPython program: at the RPython level, it contains a graph of GcStructs, with GC headers, and sometimes fields pointing at each other for the purpose of keeping the necessary GcStructs alive. At the same time, *inside* these GcStructs, there is a part that looks exactly like a C data structure. This part can contain pointers to other C data structures, which may live within other GcStructs. The C libraries will only see and modify the C data that is embedded in the "c_data" substructure of GcStructs. This is somewhat similar to the idea that if you call malloc() in a C program, you actually get a block of memory that is embedded in some larger block with malloc-specific headers; but at the level of C it doesn't matter because you only pass around and store pointers to the "embedded sub-blocks" of memory, never to the whole block with all its headers.

The detail of the memory layout for each type of ctypes object can be looked up in our on-line documentation at http://codespeak.net/pypy/dist/pypy/doc/rctypes.html#rtyping-memory-layout.

3.3.4 RCtypes implemented via a pure RPython interface

As of January 2007, a second implementation of RCtypes is being developed. The basic conclusion of the work on the first implementation is that it is quite tedious to have to write code that generates detailed low-level operations for each high-level ctypes operation - and most importantly, it is not flexible enough for our purposes. The major issue is that problems were recently found in corner cases of memory management, i.e. when ctypes objects should keep other ctypes objects alive and for how long. Fixing this would involve some pervasive changes in the low-level representation of ctypes objects, which would require all the RCtypes RTyping code to be updated. Similarly, to be able to use a Moving Garbage Collector, the memory that is visible to the external C code will need to be separated from the GC-managed memory that is used for the RCtypes objects themselves; this would also require a complete upgrade of all the RTyping code.

3.3.5 The rctypesobject RPython library

To solve this, we started writing a complete RPython library that offers the same functionality as ctypes, but with a more regular, RPython-compatible interface. This library is in the *pypy/rlib/rctypes/rctypesobject.py* module. All operations use either static or instance methods on classes following a structure similar to the ctypes classes. The flexibility of writing a normal RPython library allowed us to use internal support classes freely, together with data structures appropriate to the memory management needs.

This interface is more tedious to use than ctypes' native interface, but it is not meant for manual use. Instead, the translation toolchain maps uses of the usual ctypes interface in the RPython program to the more verbose but more regular rctypesobject interface.

The rctypesobject library is mostly implemented as a family of functions that build and return new classes. For example, `Primitive(Signed)` returns the class corresponding to C objects of the low-level type `Signed` (corresponding to `long` in C), `RPointer(Primitive(Signed))` is the class corresponding to the type "pointer to long" and `RVarArray(RPointer(Primitive(Signed)))` returns the class corresponding to arrays of pointers to longs.

Each of these classes expose a similar interface: the `allocate()` static method returns a new instance with its own memory storage (allocated with a separate `lltype.malloc()` as regular C memory, and automatically freed from the `__del__()` method of the instance). For example, `Primitive(Signed).allocate()` returns an instance that manages enough C memory to contain just one `long`. If `x` designates this instance, the expression `pointer(x)` allocates and returns an instance of class `RPointer(Primitive(Signed))` that manages a word of C memory, large enough to hold a pointer, and initialized to point to the previous `long` in memory.

The details of the interface can be found in the library's test file, *pypy/rlib/rctypes/test/test_rctypesobject.py*.

3.3.6 ControllerEntry

We implemented a generic way in the translation toolchain to map operations on arbitrary Python objects to calls to "glue" RPython classes and methods. This mechanism can be found in *pypy/rpython/controllerentry.py*. In short, it allows code to register, for each full Python object, class, or metaclass, a corresponding "controller" class. During translation, the controller is invoked when the Python object, class or metaclass is encountered. The annotator delegates to the controller the meaning of all operations - instantiation, attribute reading and setting, etc. By default, this delegation is performed by replacing the operation with a call to a method of the controller; in this way, the controller can be written simply as a family of RPython methods with names like:

- `new()` - called when the source RPython program tries to instantiate the full Python class
- `get_xyz()` and `set_xyz()` - called when the RPython program tries to get or set the attribute `xyz`
- `getitem()` and `setitem()` - called when the RPython program uses the `[ ]` notation to index the full Python object

If necessary, the controller can also choose to entirely override the default annotation and rtyping behavior and insert its own. This is useful for cases where the method cannot be implemented in RPython, e.g. in the presence of a polymorphic operation that would cause the method signature to be ill-typed.

For RCtypes, we implemented controllers that map the regular ctypes objects, classes and metaclasses to classes and operations from rctypesobject. This turned out to be a good way to separate the RCtypes implementation issues from the (sometimes complicated) interpretation of ctypes' rich and irregular interface.

3.3.7 Performance

The greatest advantage of the ControllerEntry approach over the direct RTyping approach of the first RCtypes implementation is that it is higher level, providing more flexibility. This is also potentially a disadvantage: there is for example no owner/alias analysis done during annotation -- instead, all ctypes objects of a given type are implemented identically. We speculate that the general optimizations that we implemented - most notably malloc removal [D07.1] - are either good enough to remove the overhead in the common case, or can be made good enough with some more efforts.

No performance figures have been collected at this time because the ControllerEntry approach is work in progress, and does not support all RCtypes constructs so far. (Work on this has been postponed in favor of more urgent tasks within and outside WP03.)

3.4 Conclusion

The Extension Compiler is still work in progress. With it, though, we have already laid the groundwork for providing a single platform for extension module writers to develop extension modules for all Python implementations. PyPy also provides other interesting benefits over writing code directly in C: a high-level environment in which to write, test and debug extension modules.

4 Glossary of Abbreviations

The following abbreviations may be used within this document:

4.1 Technical Abbreviations:

AOP	Aspect Oriented Programming
AST	Abstract Syntax Tree
CPython	The standard Python interpreter written in C. Generally known as "Python". Available from www.python.org.
codespeak	The name of the machine where the PyPy project is hosted.
CCLP	Concurrent Constraint Logic Programming.
CPS	Continuation-Passing Style.
CSP	Constraint Satisfaction Problem.
CLI	Common Language Infrastructure.
CLR	Common Language Runtime.
docutils	The Python documentation utilities.
F/OSS	Free and Open Source Software
GC	Garbage collector.
GenC backend	The backend for the PyPy translation toolsuite that generates C code.
GenLLVM backend	The backend for the PyPy translation toolsuite that generates LLVM code.
GenCLI backend	The backend for the PyPy translation toolsuite that generates CLI code.
Graphviz	Graph visualisation software from AT&T.
IL	Intermediate Language: the native assembler-level language of the CLI virtual machine.
Jython	A version of Python written in Java.
LLVM	Low Level Virtual Machine - a compiler infrastructure available from University of Illinois at Urbana-Champaign
LOC	Lines of code.
Object Space	A library providing objects and operations between them, available to the bytecode interpreter via a well-defined API.
Pygame	A Python extension library that wraps the Simple DirectMedia Layer - a cross-platform multimedia library designed to provide fast access to the graphics framebuffer and audio device.
pypy-c	The PyPy Standard Interpreter, translated to C and then compiled to a binary executable program
ReST	reStructuredText, the plaintext markup system used by docutils.

RPython	Restricted Python; a less dynamic subset of Python in which PyPy is written.
Standard Interpreter	The subsystem of PyPy which implements the Python language. It is divided in two components: the bytecode interpreter, and the standard object space.
Standard Object Space	An object space which implements creation, access and modification of regular Python application level objects.
VM	Virtual Machine.

4.2 Partner Acronyms:

DFKI	Deutsches Forschungszentrum für künstliche Intelligenz
HHU	Heinrich Heine Universität Düsseldorf
Strakt	AB Strakt
Logilab	Logilab
CM	Change Maker
mer	merlinux GmbH
tis	Tismerysoft GmbH
Impara	Impara GmbH

References

[CLISP] GNU CLISP, http://clisp.cons.org/

[CTYPES] *The ctypes package*, Thomas Heller, http://starship.python.net/crew/theller/ctypes/

[D04.2] *Complete Python Implementation*, PyPy EU-Report, 2005

[D05.1] *Compiling Dynamic Language Implementations*, PyPy EU-Report, 2005

[D07.1] *Support for Massive Parallelism and Publish about Optimisation results, Practical Usages and Approaches for Translation Aspects*, PyPy EU-Report, 2006

[DFFI] *The Foreign Function Call Facility*, Implementation Notes for GNU CLISP, Bruno Haible, Michael Stoll and Sam Steingold, http://clisp.cons.org/impnotes/dffi.html

[LIBFFI] *The libffi Home Page*, Anthony Green, http://sources.redhat.com/libffi/

[PEP308] http://www.python.org/dev/peps/pep-0308/

[PEP343] http://www.python.org/dev/peps/pep-0343/

[PEP352] http://www.python.org/dev/peps/pep-0352/

[PEP353] http://www.python.org/dev/peps/pep-0353/

[PEP357] http://www.python.org/dev/peps/pep-0357/

[PY24] *What's New in Python 2.4*, A.M. Kuchling, Python Software Foundation, http://www.python.org/doc/2.4/whatsnew/whatsnew24.html

[PYREX] *Pyrex - a Language for Writing Python Extension Modules*, Greg Ewing, http://www.cosc.canterbury.ac.nz/greg.ewing/python/Pyrex/

[SWIG] *Simplified Wrapper and Interface Generator*, http://www.swig.org/

What's New in Python 2.4

Release 1.01

A.M. Kuchling

November 30, 2004

Python Software Foundation
Email: amk@amk.ca

Contents

This article explains the new features in Python 2.4, released in December 2004.

Python 2.4 is a medium-sized release. It doesn't introduce as many changes as the radical Python 2.2, but introduces more features than the conservative 2.3 release. The most significant new language features are function decorators and generator expressions; most other changes are to the standard library.

According to the CVS change logs, there were 481 patches applied and 502 bugs fixed between Python 2.3 and 2.4. Both figures are likely to be underestimates.

This article doesn't attempt to provide a complete specification of every single new feature, but instead provides a brief introduction to each feature. For full details, you should refer to the documentation for Python 2.4, such as the *Python Library Reference* and the *Python Reference Manual*. Often you will be referred to the PEP for a particular new feature for explanations of the implementation and design rationale.

1 PEP 218: Built-In Set Objects

Python 2.3 introduced the `sets` module. C implementations of set data types have now been added to the Python core as two new built-in types, `set(`*iterable*`)` and `frozenset(`*iterable*`)`. They provide high speed operations for membership testing, for eliminating duplicates from sequences, and for mathematical operations like unions, intersections, differences, and symmetric differences.

```
>>> a = set('abracadabra')              # form a set from a string
>>> 'z' in a                            # fast membership testing
False
>>> a                                   # unique letters in a
set(['a', 'r', 'b', 'c', 'd'])
>>> ''.join(a)                          # convert back into a string
'arbcd'

>>> b = set('alacazam')                 # form a second set
>>> a - b                               # letters in a but not in b
set(['r', 'd', 'b'])
>>> a | b                               # letters in either a or b
set(['a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'])
>>> a & b                               # letters in both a and b
set(['a', 'c'])
>>> a ^ b                               # letters in a or b but not both
set(['r', 'd', 'b', 'm', 'z', 'l'])

>>> a.add('z')                          # add a new element
>>> a.update('wxy')                     # add multiple new elements
>>> a
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'x', 'z'])
>>> a.remove('x')                       # take one element out
>>> a
set(['a', 'c', 'b', 'd', 'r', 'w', 'y', 'z'])
```

The `frozenset` type is an immutable version of `set`. Since it is immutable and hashable, it may be used as a dictionary key or as a member of another set.

The `sets` module remains in the standard library, and may be useful if you wish to subclass the `Set` or `ImmutableSet` classes. There are currently no plans to deprecate the module.

See Also:

PEP 218, "*Adding a Built-In Set Object Type*"
Originally proposed by Greg Wilson and ultimately implemented by Raymond Hettinger.

2 PEP 237: Unifying Long Integers and Integers

The lengthy transition process for this PEP, begun in Python 2.2, takes another step forward in Python 2.4. In 2.3, certain integer operations that would behave differently after int/long unification triggered `FutureWarning` warnings and returned values limited to 32 or 64 bits (depending on your platform). In 2.4, these expressions no longer produce a warning and instead produce a different result that's usually a long integer.

The problematic expressions are primarily left shifts and lengthy hexadecimal and octal constants. For example, `2 << 32` results in a warning in 2.3, evaluating to 0 on 32-bit platforms. In Python 2.4, this expression now returns the correct answer, 8589934592.

See Also:

PEP 237, "*Unifying Long Integers and Integers*"
Original PEP written by Moshe Zadka and GvR. The changes for 2.4 were implemented by Kalle Svensson.

3 PEP 289: Generator Expressions

The iterator feature introduced in Python 2.2 and the `itertools` module make it easier to write programs that loop through large data sets without having the entire data set in memory at one time. List comprehensions don't fit into this picture very well because they produce a Python list object containing all of the items. This unavoidably pulls all of the objects into memory, which can be a problem if your data set is very large. When trying to write a functionally-styled program, it would be natural to write something like:

```
links = [link for link in get_all_links() if not link.followed]
for link in links:
    ...
```

instead of

```
for link in get_all_links():
    if link.followed:
        continue
    ...
```

The first form is more concise and perhaps more readable, but if you're dealing with a large number of link objects you'd have to write the second form to avoid having all link objects in memory at the same time.

Generator expressions work similarly to list comprehensions but don't materialize the entire list; instead they create a generator that will return elements one by one. The above example could be written as:

```
links = (link for link in get_all_links() if not link.followed)
for link in links:
    ...
```

Generator expressions always have to be written inside parentheses, as in the above example. The parentheses signalling a function call also count, so if you want to create a iterator that will be immediately passed to a function you could write:

```
print sum(obj.count for obj in list_all_objects())
```

Generator expressions differ from list comprehensions in various small ways. Most notably, the loop variable (*obj* in the above example) is not accessible outside of the generator expression. List comprehensions leave the variable assigned to its last value; future versions of Python will change this, making list comprehensions match generator

expressions in this respect.

See Also:

PEP 289, "*Generator Expressions*"
Proposed by Raymond Hettinger and implemented by Jiwon Seo with early efforts steered by Hye-Shik Chang.

4 PEP 292: Simpler String Substitutions

Some new classes in the standard library provide an alternative mechanism for substituting variables into strings; this style of substitution may be better for applications where untrained users need to edit templates.

The usual way of substituting variables by name is the % operator:

```
>>> '%(page)i: %(title)s' % {'page':2, 'title': 'The Best of Times'}
'2: The Best of Times'
```

When writing the template string, it can be easy to forget the '`i`' or '`s`' after the closing parenthesis. This isn't a big problem if the template is in a Python module, because you run the code, get an "Unsupported format character" `ValueError`, and fix the problem. However, consider an application such as Mailman where template strings or translations are being edited by users who aren't aware of the Python language. The format string's syntax is complicated to explain to such users, and if they make a mistake, it's difficult to provide helpful feedback to them.

PEP 292 adds a `Template` class to the `string` module that uses '`$`' to indicate a substitution. `Template` is a subclass of the built-in Unicode type, so the result is always a Unicode string:

```
>>> import string
>>> t = string.Template('$page: $title')
>>> t.substitute({'page':2, 'title': 'The Best of Times'})
u'2: The Best of Times'
```

If a key is missing from the dictionary, the `substitute` method will raise a `KeyError`. There's also a `safe_substitute` method that ignores missing keys:

```
>>> t = string.SafeTemplate('$page: $title')
>>> t.safe_substitute({'page':3})
u'3: $title'
```

See Also:

PEP 292, "*Simpler String Substitutions*"
Written and implemented by Barry Warsaw.

5 PEP 318: Decorators for Functions and Methods

Python 2.2 extended Python's object model by adding static methods and class methods, but it didn't extend Python's syntax to provide any new way of defining static or class methods. Instead, you had to write a `def` statement in the usual way, and pass the resulting method to a `staticmethod()` or `classmethod()` function that would wrap up the function as a method of the new type. Your code would look like this:

```
class C:
   def meth (cls):
       ...

   meth = classmethod(meth)   # Rebind name to wrapped-up class method
```

If the method was very long, it would be easy to miss or forget the `classmethod()` invocation after the function body.

The intention was always to add some syntax to make such definitions more readable, but at the time of 2.2's release a good syntax was not obvious. Today a good syntax *still* isn't obvious but users are asking for easier access to the feature; a new syntactic feature has been added to meet this need.

The new feature is called "function decorators". The name comes from the idea that `classmethod`, `staticmethod`, and friends are storing additional information on a function object; they're *decorating* functions with more details.

The notation borrows from Java and uses the '`@`' character as an indicator. Using the new syntax, the example above would be written:

```
class C:

   @classmethod
   def meth (cls):
       ...
```

The `@classmethod` is shorthand for the `meth=classmethod(meth)` assignment. More generally, if you have the following:

```
@A @B @C
def f ():
    ...
```

It's equivalent to the following pre-decorator code:

```
def f(): ...
f = A(B(C(f)))
```

Decorators must come on the line before a function definition, and can't be on the same line, meaning that `@A def f(): ...` is illegal. You can only decorate function definitions, either at the module level or inside a class; you can't decorate class definitions.

A decorator is just a function that takes the function to be decorated as an argument and returns either the same function or some new callable thing. It's easy to write your own decorators. The following simple example just sets an attribute on the function object:

```
>>> def deco(func):
...     func.attr = 'decorated'
...     return func
...
>>> @deco
... def f(): pass
...
>>> f
<function f at 0x402ef0d4>
>>> f.attr
'decorated'
>>>
```

As a slightly more realistic example, the following decorator checks that the supplied argument is an integer:

```
def require_int (func):
    def wrapper (arg):
        assert isinstance(arg, int)
        return func(arg)

    return wrapper

@require_int
def p1 (arg):
    print arg

@require_int
def p2(arg):
    print arg*2
```

An example in PEP 318 contains a fancier version of this idea that lets you both specify the required type and check the returned type.

Decorator functions can take arguments. If arguments are supplied, your decorator function is called with only those arguments and must return a new decorator function; this function must take a single function and return a function, as previously described. In other words, `@A @B @C(args)` becomes:

```
def f(): ...
_deco = C(args)
f = A(B(_deco(f)))
```

Getting this right can be slightly brain-bending, but it's not too difficult.

A small related change makes the `func_name` attribute of functions writable. This attribute is used to display function names in tracebacks, so decorators should change the name of any new function that's constructed and returned.

See Also:

PEP 318, "*Decorators for Functions, Methods and Classes*"

Written by Kevin D. Smith, Jim Jewett, and Skip Montanaro. Several people wrote patches implementing function decorators, but the one that was actually checked in was patch #979728, written by Mark Russell.

6 PEP 322: Reverse Iteration

A new built-in function, `reversed(`*seq*`)`, takes a sequence and returns an iterator that loops over the elements of the sequence in reverse order.

```
>>> for i in reversed(xrange(1,4)):
...    print i
...
3
2
1
```

Compared to extended slicing, such as `range(1,4)[::-1]`, `reversed()` is easier to read, runs faster, and uses substantially less memory.

Note that `reversed()` only accepts sequences, not arbitrary iterators. If you want to reverse an iterator, first convert it to a list with `list()`.

```
>>> input = open('/etc/passwd', 'r')
>>> for line in reversed(list(input)):
...   print line
...
root:*:0:0:System Administrator:/var/root:/bin/tcsh
  ...
```

See Also:

PEP 322, "*Reverse Iteration*"
: Written and implemented by Raymond Hettinger.

7 PEP 324: New subprocess Module

The standard library provides a number of ways to execute a subprocess, offering different features and different levels of complexity. `os.system(`*command*`)` is easy to use, but slow (it runs a shell process which executes the command) and dangerous (you have to be careful about escaping the shell's metacharacters). The `popen2` module offers classes that can capture standard output and standard error from the subprocess, but the naming is confusing. The `subprocess` module cleans this up, providing a unified interface that offers all the features you might need.

Instead of `popen2`'s collection of classes, `subprocess` contains a single class called `Popen` whose constructor supports a number of different keyword arguments.

```
class Popen(args, bufsize=0, executable=None,
    stdin=None, stdout=None, stderr=None,
    preexec_fn=None, close_fds=False, shell=False,
    cwd=None, env=None, universal_newlines=False,
    startupinfo=None, creationflags=0):
```

args is commonly a sequence of strings that will be the arguments to the program executed as the subprocess. (If the *shell* argument is true, *args* can be a string which will then be passed on to the shell for interpretation, just as `os.system()` does.)

stdin, *stdout*, and *stderr* specify what the subprocess's input, output, and error streams will be. You can provide a file object or a file descriptor, or you can use the constant `subprocess.PIPE` to create a pipe between the subprocess and the parent.

The constructor has a number of handy options:

- *close_fds* requests that all file descriptors be closed before running the subprocess.
- *cwd* specifies the working directory in which the subprocess will be executed (defaulting to whatever the parent's working directory is).
- *env* is a dictionary specifying environment variables.
- *preexec_fn* is a function that gets called before the child is started.
- *universal_newlines* opens the child's input and output using Python's universal newline feature.

Once you've created the `Popen` instance, you can call its `wait()` method to pause until the subprocess has exited, `poll()` to check if it's exited without pausing, or `communicate(`*data*`)` to send the string *data* to the subprocess's standard input. `communicate(`*data*`)` then reads any data that the subprocess has sent to its standard output or standard error, returning a tuple `(`*stdout_data*`, `*stderr_data*`)`.

`call()` is a shortcut that passes its arguments along to the `Popen` constructor, waits for the command to complete, and returns the status code of the subprocess. It can serve as a safer analog to `os.system()`:

```
sts = subprocess.call(['dpkg', '-i', '/tmp/new-package.deb'])
if sts == 0:
    # Success
    ...
else:
    # dpkg returned an error
    ...
```

The command is invoked without use of the shell. If you really do want to use the shell, you can add `shell=True` as a keyword argument and provide a string instead of a sequence:

```
sts = subprocess.call('dpkg -i /tmp/new-package.deb', shell=True)
```

The PEP takes various examples of shell and Python code and shows how they'd be translated into Python code that uses `subprocess`. Reading this section of the PEP is highly recommended.

See Also:

PEP 324, "*subprocess - New process module*"
Written and implemented by Peter Åstrand, with assistance from Fredrik Lundh and others.

8 PEP 327: Decimal Data Type

Python has always supported floating-point (FP) numbers, based on the underlying C `double` type, as a data type. However, while most programming languages provide a floating-point type, many people (even programmers) are unaware that floating-point numbers don't represent certain decimal fractions accurately. The new `Decimal` type can represent these fractions accurately, up to a user-specified precision limit.

8.1 Why is Decimal needed?

The limitations arise from the representation used for floating-point numbers. FP numbers are made up of three components:

- The sign, which is positive or negative.
- The mantissa, which is a single-digit binary number followed by a fractional part. For example, `1.01` in base-2 notation is `1 + 0/2 + 1/4`, or 1.25 in decimal notation.

- The exponent, which tells where the decimal point is located in the number represented.

For example, the number 1.25 has positive sign, a mantissa value of 1.01 (in binary), and an exponent of 0 (the decimal point doesn't need to be shifted). The number 5 has the same sign and mantissa, but the exponent is 2 because the mantissa is multiplied by 4 (2 to the power of the exponent 2); 1.25 * 4 equals 5.

Modern systems usually provide floating-point support that conforms to a standard called IEEE 754. C's `double` type is usually implemented as a 64-bit IEEE 754 number, which uses 52 bits of space for the mantissa. This means that numbers can only be specified to 52 bits of precision. If you're trying to represent numbers whose expansion repeats endlessly, the expansion is cut off after 52 bits. Unfortunately, most software needs to produce output in base 10, and common fractions in base 10 are often repeating decimals in binary. For example, 1.1 decimal is binary `1.0001100110011 ...`; .1 = 1/16 + 1/32 + 1/256 plus an infinite number of additional terms. IEEE 754 has to chop off that infinitely repeated decimal after 52 digits, so the representation is slightly inaccurate.

Sometimes you can see this inaccuracy when the number is printed:

```
>>> 1.1
1.1000000000000001
```

The inaccuracy isn't always visible when you print the number because the FP-to-decimal-string conversion is provided by the C library, and most C libraries try to produce sensible output. Even if it's not displayed, however, the inaccuracy is still there and subsequent operations can magnify the error.

For many applications this doesn't matter. If I'm plotting points and displaying them on my monitor, the difference between 1.1 and 1.1000000000000001 is too small to be visible. Reports often limit output to a certain number of decimal places, and if you round the number to two or three or even eight decimal places, the error is never apparent. However, for applications where it does matter, it's a lot of work to implement your own custom arithmetic routines.

Hence, the `Decimal` type was created.

8.2 The `Decimal` type

A new module, `decimal`, was added to Python's standard library. It contains two classes, `Decimal` and `Context`. `Decimal` instances represent numbers, and `Context` instances are used to wrap up various settings such as the precision and default rounding mode.

`Decimal` instances are immutable, like regular Python integers and FP numbers; once it's been created, you can't change the value an instance represents. `Decimal` instances can be created from integers or strings:

```
>>> import decimal
>>> decimal.Decimal(1972)
Decimal("1972")
>>> decimal.Decimal("1.1")
Decimal("1.1")
```

You can also provide tuples containing the sign, the mantissa represented as a tuple of decimal digits, and the exponent:

```
>>> decimal.Decimal((1, (1, 4, 7, 5), -2))
Decimal("-14.75")
```

Cautionary note: the sign bit is a Boolean value, so 0 is positive and 1 is negative.

Converting from floating-point numbers poses a bit of a problem: should the FP number representing 1.1 turn into the decimal number for exactly 1.1, or for 1.1 plus whatever inaccuracies are introduced? The decision was to

dodge the issue and leave such a conversion out of the API. Instead, you should convert the floating-point number into a string using the desired precision and pass the string to the `Decimal` constructor:

```
>>> f = 1.1
>>> decimal.Decimal(str(f))
Decimal("1.1")
>>> decimal.Decimal('%.12f' % f)
Decimal("1.100000000000")
```

Once you have `Decimal` instances, you can perform the usual mathematical operations on them. One limitation: exponentiation requires an integer exponent:

```
>>> a = decimal.Decimal('35.72')
>>> b = decimal.Decimal('1.73')
>>> a+b
Decimal("37.45")
>>> a-b
Decimal("33.99")
>>> a*b
Decimal("61.7956")
>>> a/b
Decimal("20.647398843930635838150289 02")
>>> a ** 2
Decimal("1275.9184")
>>> a**b
Traceback (most recent call last):
  ...
decimal.InvalidOperation: x ** (non-integer)
```

You can combine `Decimal` instances with integers, but not with floating-point numbers:

```
>>> a + 4
Decimal("39.72")
>>> a + 4.5
Traceback (most recent call last):
  ...
TypeError: You can interact Decimal only with int, long or Decimal data types.
>>>
```

`Decimal` numbers can be used with the `math` and `cmath` modules, but note that they'll be immediately converted to floating-point numbers before the operation is performed, resulting in a possible loss of precision and accuracy. You'll also get back a regular floating-point number and not a `Decimal`.

```
>>> import math, cmath
>>> d = decimal.Decimal('123456789012.345')
>>> math.sqrt(d)
351364.18288201344
>>> cmath.sqrt(-d)
351364.18288201344j
```

`Decimal` instances have a `sqrt()` method that returns a `Decimal`, but if you need other things such as trigonometric functions you'll have to implement them.

```
>>> d.sqrt()
Decimal("351364.1828820134592177245001")
```

8.3 The `Context` type

Instances of the `Context` class encapsulate several settings for decimal operations:

- `prec` is the precision, the number of decimal places.
- `rounding` specifies the rounding mode. The `decimal` module has constants for the various possibilities: `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_HALF_EVEN`, and various others.
- `traps` is a dictionary specifying what happens on encountering certain error conditions: either an exception is raised or a value is returned. Some examples of error conditions are division by zero, loss of precision, and overflow.

There's a thread-local default context available by calling `getcontext()`; you can change the properties of this context to alter the default precision, rounding, or trap handling. The following example shows the effect of changing the precision of the default context:

```
>>> decimal.getcontext().prec
28
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.14285714285714285714285714429")
>>> decimal.getcontext().prec = 9
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857143")
```

The default action for error conditions is selectable; the module can either return a special value such as infinity or not-a-number, or exceptions can be raised:

```
>>> decimal.Decimal(1) / decimal.Decimal(0)
Traceback (most recent call last):
  ...
decimal.DivisionByZero: x / 0
>>> decimal.getcontext().traps[decimal.DivisionByZero] = False
>>> decimal.Decimal(1) / decimal.Decimal(0)
Decimal("Infinity")
>>>
```

The `Context` instance also has various methods for formatting numbers such as `to_eng_string()` and `to_sci_string()`.

For more information, see the documentation for the `decimal` module, which includes a quick-start tutorial and a reference.

See Also:

PEP 327, "*Decimal Data Type*"
: Written by Facundo Batista and implemented by Facundo Batista, Eric Price, Raymond Hettinger, Aahz, and Tim Peters.

http://research.microsoft.com/~hollasch/cgindex/coding/ieeefloat.html
: A more detailed overview of the IEEE-754 representation.

http://www.lahey.com/float.htm
: The article uses Fortran code to illustrate many of the problems that floating-point inaccuracy can cause.

http://www2.hursley.ibm.com/decimal/
: A description of a decimal-based representation. This representation is being proposed as a standard, and underlies the new Python decimal type. Much of this material was written by Mike Cowlishaw, designer of the Rexx language.

9 PEP 328: Multi-line Imports

One language change is a small syntactic tweak aimed at making it easier to import many names from a module. In a `from` *module* `import` *names* statement, *names* is a sequence of names separated by commas. If the sequence is very long, you can either write multiple imports from the same module, or you can use backslashes to escape the line endings like this:

```
from SimpleXMLRPCServer import SimpleXMLRPCServer,\
            SimpleXMLRPCRequestHandler,\
            CGIXMLRPCRequestHandler,\
            resolve_dotted_attribute
```

The syntactic change in Python 2.4 simply allows putting the names within parentheses. Python ignores newlines within a parenthesized expression, so the backslashes are no longer needed:

```
from SimpleXMLRPCServer import (SimpleXMLRPCServer,
                                SimpleXMLRPCRequestHandler,
                                CGIXMLRPCRequestHandler,
                                resolve_dotted_attribute)
```

The PEP also proposes that all `import` statements be absolute imports, with a leading '`.`' character to indicate a relative import. This part of the PEP is not yet implemented, and will have to wait for Python 2.5 or some other future version.

See Also:

PEP 328, "*Imports: Multi-Line and Absolute/Relative*"
Written by Aahz. Multi-line imports were implemented by Dima Dorfman.

10 PEP 331: Locale-Independent Float/String Conversions

The `locale` modules lets Python software select various conversions and display conventions that are localized to a particular country or language. However, the module was careful to not change the numeric locale because various functions in Python's implementation required that the numeric locale remain set to the `'C'` locale. Often this was because the code was using the C library's `atof()` function.

Not setting the numeric locale caused trouble for extensions that used third-party C libraries, however, because they wouldn't have the correct locale set. The motivating example was GTK+, whose user interface widgets weren't displaying numbers in the current locale.

The solution described in the PEP is to add three new functions to the Python API that perform ASCII-only conversions, ignoring the locale setting:

- `PyOS_ascii_strtod(`*str*`, `*ptr*`)` and `PyOS_ascii_atof(`*str*`, `*ptr*`)` both convert a string to a C `double`.
- `PyOS_ascii_formatd(`*buffer*`, `*buf_len*`, `*format*`, `*d*`)` converts a `double` to an ASCII string.

The code for these functions came from the GLib library (http://developer.gnome.org/arch/gtk/glib.html), whose developers kindly relicensed the relevant functions and donated them to the Python Software Foundation. The `locale` module can now change the numeric locale, letting extensions such as GTK+ produce the correct results.

See Also:

PEP 331, "*Locale-Independent Float/String Conversions*"
Written by Christian R. Reis, and implemented by Gustavo Carneiro.

11 Other Language Changes

Here are all of the changes that Python 2.4 makes to the core Python language.

- Decorators for functions and methods were added (PEP 318).
- Built-in `set` and `frozenset` types were added (PEP 218). Other new built-ins include the `reversed(`*seq*`)` function (PEP 322).
- Generator expressions were added (PEP 289).
- Certain numeric expressions no longer return values restricted to 32 or 64 bits (PEP 237).
- You can now put parentheses around the list of names in a `from` *module* `import` *names* statement (PEP 328).
- The `dict.update()` method now accepts the same argument forms as the `dict` constructor. This includes any mapping, any iterable of key/value pairs, and keyword arguments. (Contributed by Raymond Hettinger.)
- The string methods `ljust()`, `rjust()`, and `center()` now take an optional argument for specifying a fill character other than a space. (Contributed by Raymond Hettinger.)
- Strings also gained an `rsplit()` method that works like the `split()` method but splits from the end of the string.

```
>>> 'www.python.org'.split('.', 1)
['www', 'python.org']
'www.python.org'.rsplit('.', 1)
['www.python', 'org']
```

- Three keyword parameters, *cmp*, *key*, and *reverse*, were added to the `sort()` method of lists. These parameters make some common usages of `sort()` simpler. All of these parameters are optional.

  For the *cmp* parameter, the value should be a comparison function that takes two parameters and returns -1, 0, or +1 depending on how the parameters compare. This function will then be used to sort the list. Previously this was the only parameter that could be provided to `sort()`.

  key should be a single-parameter function that takes a list element and returns a comparison key for the element. The list is then sorted using the comparison keys. The following example sorts a list case-insensitively:

```
>>> L = ['A', 'b', 'c', 'D']
>>> L.sort()                # Case-sensitive sort
>>> L
['A', 'D', 'b', 'c']
>>> # Using 'key' parameter to sort list
>>> L.sort(key=lambda x: x.lower())
>>> L
['A', 'b', 'c', 'D']
>>> # Old-fashioned way
>>> L.sort(cmp=lambda x,y: cmp(x.lower(), y.lower()))
>>> L
['A', 'b', 'c', 'D']
```

  The last example, which uses the *cmp* parameter, is the old way to perform a case-insensitive sort. It works but is slower than using a *key* parameter. Using *key* calls `lower()` method once for each element in the list while using *cmp* will call it twice for each comparison, so using *key* saves on invocations of the `lower()` method.

For simple key functions and comparison functions, it is often possible to avoid a `lambda` expression by using an unbound method instead. For example, the above case-insensitive sort is best written as:

```
>>> L.sort(key=str.lower)
>>> L
['A', 'b', 'c', 'D']
```

Finally, the *reverse* parameter takes a Boolean value. If the value is true, the list will be sorted into reverse order. Instead of `L.sort() ; L.reverse()`, you can now write `L.sort(reverse=True)`.

The results of sorting are now guaranteed to be stable. This means that two entries with equal keys will be returned in the same order as they were input. For example, you can sort a list of people by name, and then sort the list by age, resulting in a list sorted by age where people with the same age are in name-sorted order.

(All changes to `sort()` contributed by Raymond Hettinger.)

- There is a new built-in function `sorted(`*iterable*`)` that works like the in-place `list.sort()` method but can be used in expressions. The differences are:
  - the input may be any iterable;
  - a newly formed copy is sorted, leaving the original intact; and
  - the expression returns the new sorted copy

```
>>> L = [9,7,8,3,2,4,1,6,5]
>>> [10+i for i in sorted(L)]       # usable in a list comprehension
[11, 12, 13, 14, 15, 16, 17, 18, 19]
>>> L                               # original is left unchanged
[9,7,8,3,2,4,1,6,5]
>>> sorted('Monty Python')          # any iterable may be an input
[' ', 'M', 'P', 'h', 'n', 'n', 'o', 'o', 't', 't', 'y', 'y']

>>> # List the contents of a dict sorted by key values
>>> colormap = dict(red=1, blue=2, green=3, black=4, yellow=5)
>>> for k, v in sorted(colormap.iteritems()):
...     print k, v
...
black 4
blue 2
green 3
red 1
yellow 5
```

(Contributed by Raymond Hettinger.)

- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.
- The interpreter gained a new switch, **-m**, that takes a name, searches for the corresponding module on `sys.path`, and runs the module as a script. For example, you can now run the Python profiler with `python -m profile`. (Contributed by Nick Coghlan.)
- The `eval(`*expr*`, `*globals*`, `*locals*`)` and `execfile(`*filename*`, `*globals*`, `*locals*`)` functions and the `exec` statement now accept any mapping type for the *locals* parameter. Previously this had to be a regular Python dictionary. (Contributed by Raymond Hettinger.)
- The `zip()` built-in function and `itertools.izip()` now return an empty list if called with no arguments. Previously they raised a `TypeError` exception. This makes them more suitable for use with variable length argument lists:

```
>>> def transpose(array):
...    return zip(*array)
...
>>> transpose([(1,2,3), (4,5,6)])
[(1, 4), (2, 5), (3, 6)]
>>> transpose([])
[]
```

(Contributed by Raymond Hettinger.)

- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`. The incomplete module object left behind would fool further imports of the same module into succeeding, leading to confusing errors. (Fixed by Tim Peters.)
- `None` is now a constant; code that binds a new value to the name '`None`' is now a syntax error. (Contributed by Raymond Hettinger.)

11.1 Optimizations

- The inner loops for list and tuple slicing were optimized and now run about one-third faster. The inner loops for dictionaries were also optimized , resulting in performance boosts for `keys()`, `values()`, `items()`, `iterkeys()`, `itervalues()`, and `iteritems()`. (Contributed by Raymond Hettinger.)
- The machinery for growing and shrinking lists was optimized for speed and for space efficiency. Appending and popping from lists now runs faster due to more efficient code paths and less frequent use of the underlying system `realloc()`. List comprehensions also benefit. `list.extend()` was also optimized and no longer converts its argument into a temporary list before extending the base list. (Contributed by Raymond Hettinger.)
- `list()`, `tuple()`, `map()`, `filter()`, and `zip()` now run several times faster with non-sequence arguments that supply a `__len__()` method. (Contributed by Raymond Hettinger.)
- The methods `list.__getitem__()`, `dict.__getitem__()`, and `dict.__contains__()` are are now implemented as `method_descriptor` objects rather than `wrapper_descriptor` objects. This form of access doubles their performance and makes them more suitable for use as arguments to functionals: '`map(mydict.__getitem__, keylist)`'. (Contributed by Raymond Hettinger.)
- Added a new opcode, `LIST_APPEND`, that simplifies the generated bytecode for list comprehensions and speeds them up by about a third. (Contributed by Raymond Hettinger.)
- The peephole bytecode optimizer has been improved to produce shorter, faster bytecode; remarkably, the resulting bytecode is more readable. (Enhanced by Raymond Hettinger.)
- String concatenations in statements of the form `s = s + "abc"` and `s += "abc"` are now performed more efficiently in certain circumstances. This optimization won't be present in other Python implementations such as Jython, so you shouldn't rely on it; using the `join()` method of strings is still recommended when you want to efficiently glue a large number of strings together. (Contributed by Armin Rigo.)

The net result of the 2.4 optimizations is that Python 2.4 runs the pystone benchmark around 5% faster than Python 2.3 and 35% faster than Python 2.2. (pystone is not a particularly good benchmark, but it's the most commonly used measurement of Python's performance. Your own applications may show greater or smaller benefits from Python 2.4.)

12 New, Improved, and Deprecated Modules

As usual, Python's standard library received a number of enhancements and bug fixes. Here's a partial list of the most notable changes, sorted alphabetically by module name. Consult the 'Misc/NEWS' file in the source tree for a more complete list of changes, or look through the CVS logs for all the details.

- The `asyncore` module's `loop()` function now has a *count* parameter that lets you perform a limited number of passes through the polling loop. The default is still to loop forever.

- The `base64` module now has more complete RFC 3548 support for Base64, Base32, and Base16 encoding and decoding, including optional case folding and optional alternative alphabets. (Contributed by Barry Warsaw.)

- The `bisect` module now has an underlying C implementation for improved performance. (Contributed by Dmitry Vasiliev.)

- The CJKCodecs collections of East Asian codecs, maintained by Hye-Shik Chang, was integrated into 2.4. The new encodings are:

  - Chinese (PRC): gb2312, gbk, gb18030, big5hkscs, hz
  - Chinese (ROC): big5, cp950
  - Japanese: cp932, euc-jis-2004, euc-jp, euc-jisx0213, iso-2022-jp, iso-2022-jp-1, iso-2022-jp-2, iso-2022-jp-3, iso-2022-jp-ext, iso-2022-jp-2004, shift-jis, shift-jisx0213, shift-jis-2004
  - Korean: cp949, euc-kr, johab, iso-2022-kr

- Some other new encodings were added: HP Roman8, ISO_8859-11, ISO_8859-16, PCTP-154, and TIS-620.

- The UTF-8 and UTF-16 codecs now cope better with receiving partial input. Previously the `StreamReader` class would try to read more data, making it impossible to resume decoding from the stream. The `read()` method will now return as much data as it can and future calls will resume decoding where previous ones left off. (Implemented by Walter Dörwald.)

- There is a new `collections` module for various specialized collection datatypes. Currently it contains just one type, `deque`, a double-ended queue that supports efficiently adding and removing elements from either end:

```
>>> from collections import deque
>>> d = deque('ghi')        # make a new deque with three items
>>> d.append('j')           # add a new entry to the right side
>>> d.appendleft('f')       # add a new entry to the left side
>>> d                       # show the representation of the deque
deque(['f', 'g', 'h', 'i', 'j'])
>>> d.pop()                 # return and remove the rightmost item
'j'
>>> d.popleft()             # return and remove the leftmost item
'f'
>>> list(d)                 # list the contents of the deque
['g', 'h', 'i']
>>> 'h' in d                # search the deque
True
```

  Several modules, such as the `Queue` and `threading` modules, now take advantage of `collections.deque` for improved performance. (Contributed by Raymond Hettinger.)

- The `ConfigParser` classes have been enhanced slightly. The `read()` method now returns a list of the files that were successfully parsed, and the `set()` method raises `TypeError` if passed a *value* argument that isn't a string. (Contributed by John Belmonte and David Goodger.)

- The `curses` module now supports the ncurses extension `use_default_colors()`. On platforms where the terminal supports transparency, this makes it possible to use a transparent background. (Contributed by Jörg Lehmann.)

- The `difflib` module now includes an `HtmlDiff` class that creates an HTML table showing a side by side comparison of two versions of a text. (Contributed by Dan Gass.)

- The `email` package was updated to version 3.0, which dropped various deprecated APIs and removes support for Python versions earlier than 2.3. The 3.0 version of the package uses a new incremental parser for MIME messages, available in the `email.FeedParser` module. The new parser doesn't require reading the entire message into memory, and doesn't throw exceptions if a message is malformed; instead it records any problems in the `defect` attribute of the message. (Developed by Anthony Baxter, Barry Warsaw, Thomas Wouters, and others.)

- The `heapq` module has been converted to C. The resulting tenfold improvement in speed makes the module suitable for handling high volumes of data. In addition, the module has two new functions `nlargest()` and `nsmallest()` that use heaps to find the N largest or smallest values in a dataset without the expense of a full sort. (Contributed by Raymond Hettinger.)

- The `httplib` module now contains constants for HTTP status codes defined in various HTTP-related RFC documents. Constants have names such as `OK`, `CREATED`, `CONTINUE`, and `MOVED_PERMANENTLY`; use pydoc to get a full list. (Contributed by Andrew Eland.)

- The `imaplib` module now supports IMAP's THREAD command (contributed by Yves Dionne) and new `deleteacl()` and `myrights()` methods (contributed by Arnaud Mazin).

- The `itertools` module gained a `groupby(`*iterable*`[, `*func*`])` function. *iterable* is something that can be iterated over to return a stream of elements, and the optional *func* parameter is a function that takes an element and returns a key value; if omitted, the key is simply the element itself. `groupby()` then groups the elements into subsequences which have matching values of the key, and returns a series of 2-tuples containing the key value and an iterator over the subsequence.

  Here's an example to make this clearer. The *key* function simply returns whether a number is even or odd, so the result of `groupby()` is to return consecutive runs of odd or even numbers.

  ```
  >>> import itertools
  >>> L = [2, 4, 6, 7, 8, 9, 11, 12, 14]
  >>> for key_val, it in itertools.groupby(L, lambda x: x % 2):
  ...    print key_val, list(it)
  ...
  0 [2, 4, 6]
  1 [7]
  0 [8]
  1 [9, 11]
  0 [12, 14]
  >>>
  ```

  `groupby()` is typically used with sorted input. The logic for `groupby()` is similar to the UNIX `uniq` filter which makes it handy for eliminating, counting, or identifying duplicate elements:

  ```
  >>> word = 'abracadabra'
  >>> letters = sorted(word)   # Turn string into a sorted list of letters
  >>> letters
  ['a', 'a', 'a', 'a', 'a', 'b', 'b', 'c', 'd', 'r', 'r']
  >>> for k, g in itertools.groupby(letters):
  ...    print k, list(g)
  ...
  a ['a', 'a', 'a', 'a', 'a']
  b ['b', 'b']
  c ['c']
  d ['d']
  r ['r', 'r']
  >>> # List unique letters
  >>> [k for k, g in groupby(letters)]
  ['a', 'b', 'c', 'd', 'r']
  >>> # Count letter occurrences
  >>> [(k, len(list(g))) for k, g in groupby(letters)]
  [('a', 5), ('b', 2), ('c', 1), ('d', 1), ('r', 2)]
  ```

(Contributed by Hye-Shik Chang.)

- `itertools` also gained a function named `tee(`*iterator*`, `*N*`)` that returns *N* independent iterators that replicate *iterator*. If *N* is omitted, the default is 2.

```
>>> L = [1,2,3]
>>> i1, i2 = itertools.tee(L)
>>> i1,i2
(<itertools.tee object at 0x402c2080>, <itertools.tee object at 0x402c2090>)
>>> list(i1)               # Run the first iterator to exhaustion
[1, 2, 3]
>>> list(i2)               # Run the second iterator to exhaustion
[1, 2, 3]
>
```

  Note that `tee()` has to keep copies of the values returned by the iterator; in the worst case, it may need to keep all of them. This should therefore be used carefully if the leading iterator can run far ahead of the trailing iterator in a long stream of inputs. If the separation is large, then you might as well use `list()` instead. When the iterators track closely with one another, `tee()` is ideal. Possible applications include bookmarking, windowing, or lookahead iterators. (Contributed by Raymond Hettinger.)

- A number of functions were added to the `locale` module, such as `bind_textdomain_codeset()` to specify a particular encoding and a family of `l*gettext()` functions that return messages in the chosen encoding. (Contributed by Gustavo Niemeyer.)

- Some keyword arguments were added to the `logging` package's `basicConfig` function to simplify log configuration. The default behavior is to log messages to standard error, but various keyword arguments can be specified to log to a particular file, change the logging format, or set the logging level. For example:

```
import logging
logging.basicConfig(filename='/var/log/application.log',
    level=0,  # Log all messages
    format='%(levelname):%(process):%(thread):%(message)')
```

  Other additions to the `logging` package include a `log(`*level*`, `*msg*`)` convenience method, as well as a `TimedRotatingFileHandler` class that rotates its log files at a timed interval. The module already had `RotatingFileHandler`, which rotated logs once the file exceeded a certain size. Both classes derive from a new `BaseRotatingHandler` class that can be used to implement other rotating handlers.

  (Changes implemented by Vinay Sajip.)

- The `marshal` module now shares interned strings on unpacking a data structure. This may shrink the size of certain pickle strings, but the primary effect is to make '.pyc' files significantly smaller. (Contributed by Martin von Loewis.)

- The `nntplib` module's `NNTP` class gained `description()` and `descriptions()` methods to retrieve newsgroup descriptions for a single group or for a range of groups. (Contributed by Jürgen A. Erhard.)

- Two new functions were added to the `operator` module, `attrgetter(`*attr*`)` and `itemgetter(`*index*`)`. Both functions return callables that take a single argument and return the corresponding attribute or item; these callables make excellent data extractors when used with `map()` or `sorted()`. For example:

```
>>> L = [('c', 2), ('d', 1), ('a', 4), ('b', 3)]
>>> map(operator.itemgetter(0), L)
['c', 'd', 'a', 'b']
>>> map(operator.itemgetter(1), L)
[2, 1, 4, 3]
>>> sorted(L, key=operator.itemgetter(1)) # Sort list by second tuple item
[('d', 1), ('c', 2), ('b', 3), ('a', 4)]
```

(Contributed by Raymond Hettinger.)

- The `optparse` module was updated in various ways. The module now passes its messages through `gettext.gettext()`, making it possible to internationalize Optik's help and error messages. Help messages for options can now include the string `'%default'`, which will be replaced by the option's default value. (Contributed by Greg Ward.)

- The long-term plan is to deprecate the `rfc822` module in some future Python release in favor of the `email` package. To this end, the `email.Utils.formatdate()` function has been changed to make it usable as a replacement for `rfc822.formatdate()`. You may want to write new e-mail processing code with this in mind. (Change implemented by Anthony Baxter.)

- A new `urandom(`*n*`)` function was added to the `os` module, returning a string containing *n* bytes of random data. This function provides access to platform-specific sources of randomness such as '/dev/urandom' on Linux or the Windows CryptoAPI. (Contributed by Trevor Perrin.)

- Another new function: `os.path.lexists(`*path*`)` returns true if the file specified by *path* exists, whether or not it's a symbolic link. This differs from the existing `os.path.exists(`*path*`)` function, which returns false if *path* is a symlink that points to a destination that doesn't exist. (Contributed by Beni Cherniavsky.)

- A new `getsid()` function was added to the `posix` module that underlies the `os` module. (Contributed by J. Raynor.)

- The `poplib` module now supports POP over SSL. (Contributed by Hector Urtubia.)

- The `profile` module can now profile C extension functions. (Contributed by Nick Bastin.)

- The `random` module has a new method called `getrandbits(`*N*`)` that returns a long integer *N* bits in length. The existing `randrange()` method now uses `getrandbits()` where appropriate, making generation of arbitrarily large random numbers more efficient. (Contributed by Raymond Hettinger.)

- The regular expression language accepted by the `re` module was extended with simple conditional expressions, written as `(?(`*group*`)`*A*`|`*B*`)`. *group* is either a numeric group ID or a group name defined with `(?P<group>...)` earlier in the expression. If the specified group matched, the regular expression pattern *A* will be tested against the string; if the group didn't match, the pattern *B* will be used instead. (Contributed by Gustavo Niemeyer.)

- The `re` module is also no longer recursive, thanks to a massive amount of work by Gustavo Niemeyer. In a recursive regular expression engine, certain patterns result in a large amount of C stack space being consumed, and it was possible to overflow the stack. For example, if you matched a 30000-byte string of 'a' characters against the expression `(a|b)+`, one stack frame was consumed per character. Python 2.3 tried to check for stack overflow and raise a `RuntimeError` exception, but certain patterns could sidestep the checking and if you were unlucky Python could segfault. Python 2.4's regular expression engine can match this pattern without problems.

- A new `socketpair()` function, returning a pair of connected sockets, was added to the `socket` module. (Contributed by Dave Cole.)

- The `sys.exitfunc()` function has been deprecated. Code should be using the existing `atexit` module, which correctly handles calling multiple exit functions. Eventually `sys.exitfunc()` will become a purely internal interface, accessed only by `atexit`.

- The `tarfile` module now generates GNU-format tar files by default. (Contributed by Lars Gustaebel.)

- The `threading` module now has an elegantly simple way to support thread-local data. The module contains a `local` class whose attribute values are local to different threads.

```
import threading

data = threading.local()
data.number = 42
data.url = ('www.python.org', 80)
```

Other threads can assign and retrieve their own values for the `number` and `url` attributes. You can subclass `local` to initialize attributes or to add methods. (Contributed by Jim Fulton.)

- The `timeit` module now automatically disables periodic garbarge collection during the timing loop. This change makes consecutive timings more comparable. (Contributed by Raymond Hettinger.)
- The `weakref` module now supports a wider variety of objects including Python functions, class instances, sets, frozensets, deques, arrays, files, sockets, and regular expression pattern objects. (Contributed by Raymond Hettinger.)
- The `xmlrpclib` module now supports a multi-call extension for transmitting multiple XML-RPC calls in a single HTTP operation. (Contributed by Brian Quinlan.)
- The `mpz`, `rotor`, and `xreadlines` modules have been removed.

12.1 cookielib

The `cookielib` library supports client-side handling for HTTP cookies, mirroring the `Cookie` module's server-side cookie support. Cookies are stored in cookie jars; the library transparently stores cookies offered by the web server in the cookie jar, and fetches the cookie from the jar when connecting to the server. As in web browsers, policy objects control whether cookies are accepted or not.

In order to store cookies across sessions, two implementations of cookie jars are provided: one that stores cookies in the Netscape format so applications can use the Mozilla or Lynx cookie files, and one that stores cookies in the same format as the Perl libwww libary.

`urllib2` has been changed to interact with `cookielib`: `HTTPCookieProcessor` manages a cookie jar that is used when accessing URLs.

This module was contributed by John J. Lee.

12.2 doctest

The `doctest` module underwent considerable refactoring thanks to Edward Loper and Tim Peters. Testing can still be as simple as running `doctest.testmod()`, but the refactorings allow customizing the module's operation in various ways

The new `DocTestFinder` class extracts the tests from a given object's docstrings:

```
def f (x, y):
    """>>> f(2,2)
4
>>> f(3,2)
6
    """
    return x*y

finder = doctest.DocTestFinder()

# Get list of DocTest instances
tests = finder.find(f)
```

The new `DocTestRunner` class then runs individual tests and can produce a summary of the results:

```
runner = doctest.DocTestRunner()
for t in tests:
    tried, failed = runner.run(t)

runner.summarize(verbose=1)
```

The above example produces the following output:

```
1 items passed all tests:
   2 tests in f
2 tests in 1 items.
2 passed and 0 failed.
Test passed.
```

`DocTestRunner` uses an instance of the `OutputChecker` class to compare the expected output with the actual output. This class takes a number of different flags that customize its behaviour; ambitious users can also write a completely new subclass of `OutputChecker`.

The default output checker provides a number of handy features. For example, with the `doctest.ELLIPSIS` option flag, an ellipsis (‘`...`’) in the expected output matches any substring, making it easier to accommodate outputs that vary in minor ways:

```
def o (n):
    """>>> o(1)
<__main__.C instance at 0x...>
>>>
"""
```

Another special string, ‘`<BLANKLINE>`’, matches a blank line:

```
def p (n):
    """>>> p(1)
<BLANKLINE>
>>>
"""
```

Another new capability is producing a diff-style display of the output by specifying the `doctest.REPORT_UDIFF` (unified diffs), `doctest.REPORT_CDIFF` (context diffs), or `doctest.REPORT_NDIFF` (delta-style) option flags. For example:

```
def g (n):
    """>>> g(4)
here
is
a
lengthy
>>>"""
    L = 'here is a rather lengthy list of words'.split()
    for word in L[:n]:
        print word
```

Running the above function’s tests with `doctest.REPORT_UDIFF` specified, you get the following output:

```
**********************************************************************
File ''t.py'', line 15, in g
Failed example:
    g(4)
Differences (unified diff with -expected +actual):
    @@ -2,3 +2,3 @@
     is
     a
    -lengthy
    +rather
**********************************************************************
```

13 Build and C API Changes

Some of the changes to Python's build process and to the C API are:

- Three new convenience macros were added for common return values from extension functions: `Py_RETURN_NONE`, `Py_RETURN_TRUE`, and `Py_RETURN_FALSE`. (Contributed by Brett Cannon.)
- Another new macro, `Py_CLEAR(`*obj*`)`, decreases the reference count of *obj* and sets *obj* to the null pointer. (Contributed by Jim Fulton.)
- A new function, `PyTuple_Pack(`*N*`, `*obj1*`, `*obj2*`, ..., `*objN*`)`, constructs tuples from a variable length argument list of Python objects. (Contributed by Raymond Hettinger.)
- A new function, `PyDict_Contains(`*d*`, `*k*`)`, implements fast dictionary lookups without masking exceptions raised during the look-up process. (Contributed by Raymond Hettinger.)
- The `Py_IS_NAN(`*X*`)` macro returns 1 if its float or double argument *X* is a NaN. (Contributed by Tim Peters.)
- C code can avoid unnecessary locking by using the new `PyEval_ThreadsInitialized()` function to tell if any thread operations have been performed. If this function returns false, no lock operations are needed. (Contributed by Nick Coghlan.)
- A new function, `PyArg_VaParseTupleAndKeywords()`, is the same as `PyArg_ParseTupleAndKeywords()` but takes a `va_list` instead of a number of arguments. (Contributed by Greg Chapman.)
- A new method flag, `METH_COEXISTS`, allows a function defined in slots to co-exist with a `PyCFunction` having the same name. This can halve the access time for a method such as `set.__contains__()`. (Contributed by Raymond Hettinger.)
- Python can now be built with additional profiling for the interpreter itself, intended as an aid to people developing the Python core. Providing **-—enable-profiling** to the **configure** script will let you profile the interpreter with **gprof**, and providing the **-—with-tsc** switch enables profiling using the Pentium's Time-Stamp-Counter register. Note that the **-—with-tsc** switch is slightly misnamed, because the profiling feature also works on the PowerPC platform, though that processor architecture doesn't call that register "the TSC register". (Contributed by Jeremy Hylton.)
- The `tracebackobject` type has been renamed to `PyTracebackObject`.

13.1 Port-Specific Changes

- The Windows port now builds under MSVC++ 7.1 as well as version 6. (Contributed by Martin von Loewis.)

14 Porting to Python 2.4

This section lists previously described changes that may require changes to your code:

- Left shifts and hexadecimal/octal constants that are too large no longer trigger a `FutureWarning` and return a value limited to 32 or 64 bits; instead they return a long integer.
- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.
- The `zip()` built-in function and `itertools.izip()` now return an empty list instead of raising a `TypeError` exception if called with no arguments.
- `dircache.listdir()` now passes exceptions to the caller instead of returning empty lists.
- `LexicalHandler.startDTD()` used to receive the public and system IDs in the wrong order. This has been corrected; applications relying on the wrong order need to be fixed.
- `fcntl.ioctl` now warns if the *mutate* argument is omitted and relevant.
- The `tarfile` module now generates GNU-format tar files by default.
- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`.
- `None` is now a constant; code that binds a new value to the name '`None`' is now a syntax error.

15 Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Koray Can, Hye-Shik Chang, Michael Dyck, Raymond Hettinger, Brian Hurt, Hamish Lawson, Fredrik Lundh.

PyPy Extension Compiler

Contents

This document describes the PyPy extension compiler, which is able to compile a set of source code to a PyPy or a CPython extension module.

WARNING: this is beta software, APIs and details may change.

The documentation corresponds to release 0.99. The extension compiler has not been extensively tested and polished so far, so bugs and rough edges are likely.

The final version of the Technical Report D03.1 also describes the extension compiler's goals in more details and presents an overview of its implementation (http://codespeak.net/pypy/dist/pypy/doc/index-report.html).

1 Understanding the ext compiler

In regular Python, the ability for users to write external modules is of great importance. These external modules must be written in C, which is both an advantage - it allows access to external C libraries, low-level features, or raw performance - and an inconvenience. In the context of PyPy the greatest drawback of hard-coded C external modules is that low-level details like multithreading (e.g. locks) and memory management must be explicitly written in such a language, which would prevent the same module from being used in several differently-compiled versions of `pypy-c`.

PyPy provides instead a generic way to write modules for all Python implementations: the so-called *mixed module* approach. A single mixed module is implemented as a set of Python source files, making a subpackage of the pypy/module/ package. While running PyPy, each of these subpackages appears to be a single module, whose interface is specified in the `__init__.py` of the subpackage, and whose implementation is lazily loaded from the various other `.py` files of the subpackage.

The subpackage is written in a way that allows it to be reused for non-PyPy implementations of Python. The goal of the *Extension Compiler* is to compile a mixed module into an extension module for these other Python implementation (so far, this means CPython only).

This means that you can do the following with a mixed module:

- run it on top of CPython. This uses the CPy Object Space to directly run all space operations. Example:

```
$ python
>>> from pypy.interpreter.mixedmodule import testmodule
```

```
>>> demo = testmodule("_demo")
[ignore output here]
>>> demo.measuretime(1000000, long)
5
```

- compile it as a CPython extension module. Example:

```
$ python pypy/bin/compilemodule.py _demo
[lots of output]
Created '/tmp/usession-5/_demo/_demo.so'.
$ cd /tmp/usession-5/_demo
$ python
>>> import _demo
>>> _demo.measuretime(10000000, long)
2
```

- run it with PyPy on top of CPython. Example:

```
$ python pypy/bin/py.py --usemodules=_demo
PyPy 0.99.0 in StdObjSpace on top of Python 2.4.3
>>>> import _demo
>>>> _demo.measuretime(10000, long)
[ignore output here]
4
```

- compile it together with PyPy. It becomes a built-in module of `pypy-c`. Example:

```
$ cd pypy/translator/goal
$ python translate.py targetpypystandalone --usemodules=_demo
[wait for 30 minutes]
[translation:info] created: ./pypy-c
(Pbd)
$ ./pypy-c
Python 2.4.1 (pypy 0.9.0 build xxxxx) on linux2
Type "help", "copyright", "credits" or "license" for more information.
(InteractiveConsole)
>>>> import _demo
>>>> _demo.measuretime(10000000, long)
2
```

2 Using the ext compiler

2.1 Writing a module

You have to put your module into its own directory in pypy/module/ of your local pypy checkout. See the following directories as guidelines:

- pypy/module/_demo

  A demo module showing a few of the more interesting features.

- pypy/module/readline

  A tiny, in-progress example giving bindings to the GNU `readline` library.

- pypy/module/_sre

An algorithmic example: the regular expression engine of CPython, rewritten in RPython.

Modules can be based on ctypes. This is the case in the pypy/module/readline and pypy/module/_demo examples: they use ctypes to access functions in external C libraries. When translated to C, the calls in these examples become static, regular C function calls -- which means that most of the efficiency overhead of using ctypes disappears during the translation. However, some rules must be followed in order to make ctypes translatable; for more information, see the documentation of RCtypes.

All these modules are called "mixed" because they mix interpreter-level and application-level submodules to present a single, coherent module to the user. For a CPython extension module, you can think about interpreter-level as what will be compiled into C, and application-level as what will stay as it is, as Python code, included (as a frozen bytecode) within the C module. The capability to group several interpreter-level files into the final compiled module is similar to having a CPython extension module that was compiled from a set of C sources; but the capability to also include nicely integrated Python sources in the C extension module has no direct equivalent in hand-written C extension modules.

The directory structure is described in the Coding Guide, section mixed modules. The interpreter-level parts of a mixed module must follow the RPython programming style; this is necessary to allow them to be annotated and translated to C. Whenever they manipulate application-level (i.e. user-visible) objects, they must do so via object space operations (see wrapping rules).

2.2 Exporting functions and types

The interpreter-level parts of a mixed module exports functions and types to app-level, to make them visible to the module user. From the user's point of view, they are built-in functions and built-in types.

Exporting functions is done via the `interpleveldefs` dictionary in `__init__.py`. The functions thus exported must either have a simple signature of the form `(space, w_arg1, w_arg2, w_arg3...)` -- in this case, it will be exposed to app-level as a built-in function taking the same number of arguments, and calling the built-in function from app-level will pass all the provided arguments as wrapped objects to the interp-level function. Alternatively, the interp-level function can have an `unwrap_spec` attribute that declares what type of arguments it expects. The `unwrap_spec` is a list of specifiers, one per argument; the specifiers can be one of the following strings or objects (to be imported from pypy/interpreter/gateway.py):

- `ObjSpace`: used to specify the space argument.
- `W_Root`: for wrapped arguments to keep wrapped.
- `str`: check that the argument is a string, and unwrap it.
- `int`: check that the argument is an integer, and unwrap it.
- `float`: check that the argument is a float, and unwrap it.
- `"self"`: for methods of Wrappable subclasses, the unwrapped self.
- *Wrappable-subcls*: an instance of the given Wrappable subclass, unwrapped.
- `Arguments`: an instance of `pypy.interpreter.argument.Arguments` collecting the rest of the positional and keyword arguments.
- `"args_w"`: the rest of the positional arguments as list of wrapped objects.
- `"w_args"`: the rest of the positional arguments as a single wrapped tuple.
- *(function, cls)*: use function to check/unwrap argument of type cls.

There are two ways to export types. The first is to write the type at app-level, as a regular class. You can then call interp-level functions from your own module to implement some of the methods, while keeping the class generally at app-level.

This does not work, however, if you need special data attached to the instances of your class. For this case, you need the second solution: write the class entirely at interp-level. Such a class must inherit from `pypy.interpreter.baseobjspace.Wrappable`. You can manipulate instances of such a class freely at interp-level. Instances of subclasses of `Wrappable` are *not* wrapped; they are merely wrappable. To expose them to app-level, call `w_obj = space.wrap(obj)` -- it wraps your instance `obj` into a box, which can be passed to app-level. To unwrap such a thing again, call `obj = space.interp_w(YourSubclass, w_obj)`, which performs a type check and returns the unwrapped instance of `YourSubclass`. To avoid confusion, try to follow the usual naming convension: non-wrapped objects like instances of `Wrappable` go in variables whose name does *not* start with `w_`; wrapped objects go in variables whose name starts with `w_`.

Before you can wrap instances of subclasses of `Wrappable`, though, you need to attach a `TypeDef` to the subclass in question. The `TypeDef` describes the public app-level interface of your type. An example can be found in pypy/module/_sre/interp_sre.py. Some minimal documentation is provided here, but note that the extcompiler itself does not support special methods at the moment, i.e. methods with `__xyz__()` kind of names. You can only export regular methods (with `interp2app()`), properties (with `GetSetProperty()`), and simple class-level attributes like strings and integers. (This limitation will be lifted soon.)

2.3 Testing a module

The readline mixed module has a minimal example for each of 4 different kind of tests:

- As this is a ctypes bindings module, we should test the ctypes bindings directly to see if they work as expected:

  ```
  python pypy/test_all.py pypy/module/readline/test/test_c_readline.py
  ```

- We should then test that the mixed module wrapping is correct, using the helper function `pypy.interpreter.mixedmodule.testmodule()`. Called from a normal CPython program, this function exposes a mixed module as a plain CPython module for testing and inspection (it works in the interactive interpreter too, of course: it is a good way to see how mixed module wrappings look like in the end):

  ```
  python pypy/test_all.py pypy/module/readline/test/test_mixedmodule.py
  ```

- We can also run the mixed module within PyPy, on top of CPython. To do so from a test, we use a special `AppTestXyz` class:

  ```
  python pypy/test_all.py pypy/module/readline/test/test_with_pypy.py
  ```

- Finally, we can compile the mixed module to a CPython extension module, re-import it into the running CPython interpreter, and test it. Only this test will pick up the translation failures caused by breaking the RPython rules. (To debug translation failures, though, you should use `compilemodule.py` as described below: you will then get a Pdb prompt and a flow graph viewer to look around.)

  ```
  python pypy/test_all.py pypy/module/readline/test/test_compiler.py
  ```

2.4 Translating a module to a CPython extension

As seen in the introduction, you translate a module into a CPython extension with the following command-line:

```
python pypy/bin/compilemodule.py _demo
```

The extension compiler imports the specified package from pypy/module/ and produces a shared library importable from your local Python installation. The produced shared library is put into a temporary directory printed at the end (which on Linux is also accessible as */tmp/usession-<username>/<modulename>/<mod*

Note that we recommend you to write and run tests for your module first. This is not only a matter of style: bogus modules are likely to make the translation tool-chain fail in mysterious ways.

See the introduction for other things you can do with a mixed module.

Note that you obviously need to have a full pypy checkout first. If you have troubles compiling the demo modules, check out our ctypes-specific installation notes.