What's New in Python 2.5

Release 1.01

A.M. Kuchling

December 19, 2008

amk@amk.ca

Contents

This article explains the new features in Python 2.5. The final release of Python 2.5 is scheduled for August 2006; PEP 356 describes the planned release schedule.

The changes in Python 2.5 are an interesting mix of language and library improvements. The library enhancements will be more important to Python's user community, I think, because several widely-useful packages were added. New modules include ElementTree for XML processing (section 13.2), the SQLite database module (section 13.4), and the `ctypes` module for calling C functions (section 13.1).

The language changes are of middling significance. Some pleasant new features were added, but most of them aren't features that you'll use every day. Conditional expressions were finally added to the language using a novel syntax; see section 1. The new '`with`' statement will make writing cleanup code easier (section 8). Values can now be passed into generators (section 7). Imports are now visible as either absolute or relative (section 4). Some corner cases of exception handling are handled better (section 6). All these improvements are worthwhile, but they're improvements to one specific language feature or another; none of them are broad modifications to Python's semantics.

As well as the language and library additions, other improvements and bugfixes were made throughout the source tree. A search through the SVN change logs finds there were 353 patches applied and 458 bugs fixed between Python 2.4 and 2.5. (Both figures are likely to be underestimates.)

This article doesn't try to be a complete specification of the new features; instead changes are briefly introduced using helpful examples. For full details, you should always refer to the documentation for Python 2.5 at http://docs.python.org. If you want to understand the complete implementation and design rationale, refer to the PEP for a particular new feature.

Comments, suggestions, and error reports for this document are welcome; please e-mail them to the author or open a bug in the Python bug tracker.

1 PEP 308: Conditional Expressions

For a long time, people have been requesting a way to write conditional expressions, which are expressions that return value A or value B depending on whether a Boolean value is true or false. A conditional expression lets you write a single assignment statement that has the same effect as the following:

```
if condition:
    x = true_value
else:
    x = false_value
```

There have been endless tedious discussions of syntax on both python-dev and comp.lang.python. A vote was even held that found the majority of voters wanted conditional expressions in some form, but there was no syntax that was preferred by a clear majority. Candidates included C's `cond ? true_v : false_v`, `if cond then true_v else false_v`, and 16 other variations.

Guido van Rossum eventually chose a surprising syntax:

```
x = true_value if condition else false_value
```

Evaluation is still lazy as in existing Boolean expressions, so the order of evaluation jumps around a bit. The *condition* expression in the middle is evaluated first, and the *true_value* expression is evaluated only if the condition was true. Similarly, the *false_value* expression is only evaluated when the condition is false.

This syntax may seem strange and backwards; why does the condition go in the *middle* of the expression, and not in the front as in C's `c ? x : y`? The decision was checked by applying the new syntax to the modules in the standard library and seeing how the resulting code read. In many cases where a conditional expression is used, one value seems to be the 'common case' and one value is an 'exceptional case', used only on rarer occasions when the condition isn't met. The conditional syntax makes this pattern a bit more obvious:

```
contents = ((doc + '\n') if doc else '')
```

I read the above statement as meaning "here *contents* is usually assigned a value of `doc+'\n'`; sometimes *doc* is empty, in which special case an empty string is returned." I doubt I will use conditional expressions very often where there isn't a clear common and uncommon case.

There was some discussion of whether the language should require surrounding conditional expressions with parentheses. The decision was made to *not* require parentheses in the Python language's grammar, but as a matter of style I think you should always use them. Consider these two statements:

```
# First version -- no parens
level = 1 if logging else 0

# Second version -- with parens
level = (1 if logging else 0)
```

In the first version, I think a reader's eye might group the statement into 'level = 1', 'if logging', 'else 0', and think that the condition decides whether the assignment to *level* is performed. The second version reads better, in my opinion, because it makes it clear that the assignment is always performed and the choice is being made between two values.

Another reason for including the brackets: a few odd combinations of list comprehensions and lambdas could look like incorrect conditional expressions. See PEP 308 for some examples. If you put parentheses around your conditional expressions, you won't run into this case.

See Also:

PEP 308, "*Conditional Expressions*"
PEP written by Guido van Rossum and Raymond D. Hettinger; implemented by Thomas Wouters.

2 PEP 309: Partial Function Application

The `functools` module is intended to contain tools for functional-style programming.

One useful tool in this module is the `partial()` function. For programs written in a functional style, you'll sometimes want to construct variants of existing functions that have some of the parameters filled in. Consider a Python function `f(a, b, c)`; you could create a new function `g(b, c)` that was equivalent to `f(1, b, c)`. This is called "partial function application".

`partial` takes the arguments `(`*function*`,` *arg1*`,` *arg2*`, ...` *kwarg1=value1*`,` *kwarg2=value2*`)`. The resulting object is callable, so you can just call it to invoke *function* with the filled-in arguments.

Here's a small but realistic example:

```
import functools

def log (message, subsystem):
    "Write the contents of 'message' to the specified subsystem."
    print '%s: %s' % (subsystem, message)
    ...

server_log = functools.partial(log, subsystem='server')
server_log('Unable to open socket')
```

Here's another example, from a program that uses PyGTK. Here a context-sensitive pop-up menu is being constructed dynamically. The callback provided for the menu option is a partially applied version of the `open_item()` method, where the first argument has been provided.

```
...
class Application:
    def open_item(self, path):
       ...
    def init (self):
        open_func = functools.partial(self.open_item, item_path)
        popup_menu.append( ("Open", open_func, 1) )
```

Another function in the `functools` module is the `update_wrapper(`*wrapper*`, `*wrapped*`)` function that helps you write well-behaved decorators. `update_wrapper()` copies the name, module, and docstring attribute to a wrapper function so that tracebacks inside the wrapped function are easier to understand. For example, you might write:

```
def my_decorator(f):
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    functools.update_wrapper(wrapper, f)
    return wrapper
```

`wraps()` is a decorator that can be used inside your own decorators to copy the wrapped function's information. An alternate version of the previous example would be:

```
def my_decorator(f):
    @functools.wraps(f)
    def wrapper(*args, **kwds):
        print 'Calling decorated function'
        return f(*args, **kwds)
    return wrapper
```

See Also:

PEP 309, "*Partial Function Application*"
: PEP proposed and written by Peter Harris; implemented by Hye-Shik Chang and Nick Coghlan, with adaptations by Raymond Hettinger.

3 PEP 314: Metadata for Python Software Packages v1.1

Some simple dependency support was added to Distutils. The `setup()` function now has `requires`, `provides`, and `obsoletes` keyword parameters. When you build a source distribution using the `sdist` command, the dependency information will be recorded in the 'PKG-INFO' file.

Another new keyword parameter is `download_url`, which should be set to a URL for the package's source code. This means it's now possible to look up an entry in the package index, determine the dependencies for a package, and download the required packages.

```
VERSION = '1.0'
setup(name='PyPackage',
      version=VERSION,
      requires=['numarray', 'zlib (>=1.1.4)'],
      obsoletes=['OldPackage']
      download_url=('http://www.example.com/pypackage/dist/pkg-%s.tar.gz'
                    % VERSION),
     )
```

Another new enhancement to the Python package index at http://cheeseshop.python.org is storing source and binary archives for a package. The new `upload` Distutils command will upload a package to the repository.

Before a package can be uploaded, you must be able to build a distribution using the `sdist` Distutils command. Once that works, you can run `python setup.py upload` to add your package to the PyPI archive. Optionally you can GPG-sign the package by supplying the **--sign** and **--identity** options.

Package uploading was implemented by Martin von Löwis and Richard Jones.

See Also:

PEP 314, "*Metadata for Python Software Packages v1.1*"
PEP proposed and written by A.M. Kuchling, Richard Jones, and Fred Drake; implemented by Richard Jones and Fred Drake.

4 PEP 328: Absolute and Relative Imports

The simpler part of PEP 328 was implemented in Python 2.4: parentheses could now be used to enclose the names imported from a module using the `from ... import ...` statement, making it easier to import many different names.

The more complicated part has been implemented in Python 2.5: importing a module can be specified to use absolute or package-relative imports. The plan is to move toward making absolute imports the default in future versions of Python.

Let's say you have a package directory like this:

```
pkg/
pkg/__init__.py
pkg/main.py
pkg/string.py
```

This defines a package named `pkg` containing the `pkg.main` and `pkg.string` submodules.

Consider the code in the 'main.py' module. What happens if it executes the statement `import string`? In Python 2.4 and earlier, it will first look in the package's directory to perform a relative import, finds 'pkg/string.py', imports the contents of that file as the `pkg.string` module, and that module is bound to the name '`string`' in the `pkg.main` module's namespace.

That's fine if `pkg.string` was what you wanted. But what if you wanted Python's standard `string` module? There's no clean way to ignore `pkg.string` and look for the standard module; generally you had to look at the contents of `sys.modules`, which is slightly unclean. Holger Krekel's `py.std` package provides a tidier way to perform imports from the standard library, `import py ; py.std.string.join()`, but that package isn't available on all Python installations.

Reading code which relies on relative imports is also less clear, because a reader may be confused about which module, `string` or `pkg.string`, is intended to be used. Python users soon learned not to duplicate the names of standard library modules in the names of their packages' submodules, but you can't protect against having your submodule's name being used for a new module added in a future version of Python.

In Python 2.5, you can switch `import`'s behaviour to absolute imports using a `from __future__ import absolute_import` directive. This absolute-import behaviour will become the default in a future version (probably Python 2.7). Once absolute imports are the default, `import string` will always find the standard library's version. It's suggested that users should begin using absolute imports as much as possible, so it's preferable to begin writing `from pkg import string` in your code.

Relative imports are still possible by adding a leading period to the module name when using the `from ... import` form:

```
# Import names from pkg.string
from .string import name1, name2
# Import pkg.string
from . import string
```

This imports the `string` module relative to the current package, so in `pkg.main` this will import *name1* and *name2* from `pkg.string`. Additional leading periods perform the relative import starting from the parent of the current package. For example, code in the `A.B.C` module can do:

```
from . import D                 # Imports A.B.D
from .. import E                # Imports A.E
from ..F import G               # Imports A.F.G
```

Leading periods cannot be used with the `import` *modname* form of the import statement, only the `from ... import` form.

See Also:

PEP 328, "*Imports: Multi-Line and Absolute/Relative*"
PEP written by Aahz; implemented by Thomas Wouters.

http://codespeak.net/py/current/doc/index.html
The py library by Holger Krekel, which contains the `py.std` package.

5 PEP 338: Executing Modules as Scripts

The **-m** switch added in Python 2.4 to execute a module as a script gained a few more abilities. Instead of being implemented in C code inside the Python interpreter, the switch now uses an implementation in a new module, `runpy`.

The `runpy` module implements a more sophisticated import mechanism so that it's now possible to run modules in a package such as `pychecker.checker`. The module also supports alternative import mechanisms such as the `zipimport` module. This means you can add a .zip archive's path to `sys.path` and then use the **-m** switch to execute code from the archive.

See Also:

PEP 338, "*Executing modules as scripts*"
PEP written and implemented by Nick Coghlan.

6 PEP 341: Unified try/except/finally

Until Python 2.5, the `try` statement came in two flavours. You could use a `finally` block to ensure that code is always executed, or one or more `except` blocks to catch specific exceptions. You couldn't combine both `except` blocks and a `finally` block, because generating the right bytecode for the combined version was complicated and it wasn't clear what the semantics of the combined statement should be.

Guido van Rossum spent some time working with Java, which does support the equivalent of combining `except` blocks and a `finally` block, and this clarified what the statement should mean. In Python 2.5, you can now write:

```
try:
    block-1 ...
except Exception1:
    handler-1 ...
except Exception2:
    handler-2 ...
else:
    else-block
finally:
    final-block
```

The code in *block-1* is executed. If the code raises an exception, the various `except` blocks are tested: if the exception is of class `Exception1`, *handler-1* is executed; otherwise if it's of class `Exception2`, *handler-2* is executed, and so forth. If no exception is raised, the *else-block* is executed.

No matter what happened previously, the *final-block* is executed once the code block is complete and any raised exceptions handled. Even if there's an error in an exception handler or the *else-block* and a new exception is raised, the code in the *final-block* is still run.

See Also:

PEP 341, "*Unifying try-except and try-finally*"
PEP written by Georg Brandl; implementation by Thomas Lee.

7 PEP 342: New Generator Features

Python 2.5 adds a simple way to pass values *into* a generator. As introduced in Python 2.3, generators only produce output; once a generator's code was invoked to create an iterator, there was no way to pass any new information into the function when its execution is resumed. Sometimes the ability to pass in some information would be useful. Hackish solutions to this include making the generator's code look at a global variable and then changing the global variable's value, or passing in some mutable object that callers then modify.

To refresh your memory of basic generators, here's a simple example:

```
def counter (maximum):
    i = 0
    while i < maximum:
        yield i
        i += 1
```

When you call `counter(10)`, the result is an iterator that returns the values from 0 up to 9. On encountering the `yield` statement, the iterator returns the provided value and suspends the function's execution, preserving the local variables. Execution resumes on the following call to the iterator's `next()` method, picking up after the `yield` statement.

In Python 2.3, `yield` was a statement; it didn't return any value. In 2.5, `yield` is now an expression, returning a value that can be assigned to a variable or otherwise operated on:

```
val = (yield i)
```

I recommend that you always put parentheses around a `yield` expression when you're doing something with the returned value, as in the above example. The parentheses aren't always necessary, but it's easier to always add them instead of having to remember when they're needed.

(PEP 342 explains the exact rules, which are that a `yield`-expression must always be parenthesized except when it occurs at the top-level expression on the right-hand side of an assignment. This means you can write `val = yield i` but have to use parentheses when there's an operation, as in `val = (yield i) + 12`.)

Values are sent into a generator by calling its `send(`*value*`)` method. The generator's code is then resumed and the `yield` expression returns the specified *value*. If the regular `next()` method is called, the `yield` returns `None`.

Here's the previous example, modified to allow changing the value of the internal counter.

```
def counter (maximum):
    i = 0
    while i < maximum:
        val = (yield i)
        # If value provided, change counter
        if val is not None:
            i = val
        else:
            i += 1
```

And here's an example of changing the counter:

```
>>> it = counter(10)
>>> print it.next()
0
>>> print it.next()
1
>>> print it.send(8)
8
>>> print it.next()
9
>>> print it.next()
Traceback (most recent call last):
  File ``t.py'', line 15, in ?
    print it.next()
StopIteration
```

`yield` will usually return `None`, so you should always check for this case. Don't just use its value in expressions unless you're sure that the `send()` method will be the only method used to resume your generator function.

In addition to `send()`, there are two other new methods on generators:

- `throw(`*type*`, `*value*`=None, `*traceback*`=None)` is used to raise an exception inside the generator; the exception is raised by the `yield` expression where the generator's execution is paused.
- `close()` raises a new `GeneratorExit` exception inside the generator to terminate the iteration. On receiving this exception, the generator's code must either raise `GeneratorExit` or `StopIteration`. Catching the `GeneratorExit` exception and returning a value is illegal and will trigger a `RuntimeError`; if the function raises some other exception, that exception is propagated to the caller. `close()` will also be called by Python's garbage collector when the generator is garbage-collected.

  If you need to run cleanup code when a `GeneratorExit` occurs, I suggest using a `try: ... finally:` suite instead of catching `GeneratorExit`.

The cumulative effect of these changes is to turn generators from one-way producers of information into both producers and consumers.

Generators also become *coroutines*, a more generalized form of subroutines. Subroutines are entered at one point and exited at another point (the top of the function, and a `return` statement), but coroutines can be entered, exited, and resumed at many different points (the `yield` statements). We'll have to figure out patterns for using coroutines effectively in Python.

The addition of the `close()` method has one side effect that isn't obvious. `close()` is called when a generator is garbage-collected, so this means the generator's code gets one last chance to run before the generator is destroyed. This last chance means that `try...finally` statements in generators can now be guaranteed to work; the `finally` clause will now always get a chance to run. The syntactic restriction that you couldn't mix `yield` statements with a `try...finally` suite has therefore been removed. This seems like a minor bit of language trivia, but using generators and `try...finally` is actually necessary in order to implement the `with` statement described by PEP 343. I'll look at this new statement in the following section.

Another even more esoteric effect of this change: previously, the `gi_frame` attribute of a generator was always a frame object. It's now possible for `gi_frame` to be `None` once the generator has been exhausted.

See Also:

PEP 342, "*Coroutines via Enhanced Generators*"

PEP written by Guido van Rossum and Phillip J. Eby; implemented by Phillip J. Eby. Includes examples of some fancier uses of generators as coroutines.

Earlier versions of these features were proposed in PEP 288 by Raymond Hettinger and PEP 325 by Samuele Pedroni.

http://en.wikipedia.org/wiki/Coroutine
The Wikipedia entry for coroutines.

http://www.sidhe.org/˜dan/blog/archives/000178.html
An explanation of coroutines from a Perl point of view, written by Dan Sugalski.

8 PEP 343: The 'with' statement

The '`with`' statement clarifies code that previously would use `try...finally` blocks to ensure that clean-up code is executed. In this section, I'll discuss the statement as it will commonly be used. In the next section, I'll examine the implementation details and show how to write objects for use with this statement.

The '`with`' statement is a new control-flow structure whose basic structure is:

```
with expression [as variable]:
    with-block
```

The expression is evaluated, and it should result in an object that supports the context management protocol (that is, has `__enter__()` and `__exit__()` methods.

The object's `__enter__()` is called before *with-block* is executed and therefore can run set-up code. It also may return a value that is bound to the name *variable*, if given. (Note carefully that *variable* is *not* assigned the result of *expression*.)

After execution of the *with-block* is finished, the object's `__exit__()` method is called, even if the block raised an exception, and can therefore run clean-up code.

To enable the statement in Python 2.5, you need to add the following directive to your module:

```
from __future__ import with_statement
```

The statement will always be enabled in Python 2.6.

Some standard Python objects now support the context management protocol and can be used with the '`with`' statement. File objects are one example:

```
with open('/etc/passwd', 'r') as f:
    for line in f:
        print line
        ... more processing code ...
```

After this statement has executed, the file object in *f* will have been automatically closed, even if the `for` loop raised an exception part-way through the block.

Note: In this case, *f* is the same object created by `open()`, because `file.__enter__()` returns *self*.

The `threading` module's locks and condition variables also support the '`with`' statement:

```
lock = threading.Lock()
with lock:
    # Critical section of code
    ...
```

The lock is acquired before the block is executed and always released once the block is complete.

The new `localcontext()` function in the `decimal` module makes it easy to save and restore the current decimal context, which encapsulates the desired precision and rounding characteristics for computations:

```
from decimal import Decimal, Context, localcontext

# Displays with default precision of 28 digits
v = Decimal('578')
print v.sqrt()

with localcontext(Context(prec=16)):
    # All code in this block uses a precision of 16 digits.
    # The original context is restored on exiting the block.
    print v.sqrt()
```

8.1 Writing Context Managers

Under the hood, the '`with`' statement is fairly complicated. Most people will only use '`with`' in company with existing objects and don't need to know these details, so you can skip the rest of this section if you like. Authors of new objects will need to understand the details of the underlying implementation and should keep reading.

A high-level explanation of the context management protocol is:

- The expression is evaluated and should result in an object called a "context manager". The context manager must have `__enter__()` and `__exit__()` methods.
- The context manager's `__enter__()` method is called. The value returned is assigned to *VAR*. If no `'as` *VAR*`'` clause is present, the value is simply discarded.
- The code in *BLOCK* is executed.
- If *BLOCK* raises an exception, the `__exit__(`*type*`,` *value*`,` *traceback*`)` is called with the exception details, the same values returned by `sys.exc_info()`. The method's return value controls whether the exception is re-raised: any false value re-raises the exception, and `True` will result in suppressing it. You'll only rarely want to suppress the exception, because if you do the author of the code containing the '`with`' statement will never realize anything went wrong.
- If *BLOCK* didn't raise an exception, the `__exit__()` method is still called, but *type*, *value*, and *traceback* are all `None`.

Let's think through an example. I won't present detailed code but will only sketch the methods necessary for a database that supports transactions.

(For people unfamiliar with database terminology: a set of changes to the database are grouped into a transaction. Transactions can be either committed, meaning that all the changes are written into the database, or rolled back, meaning that the changes are all discarded and the database is unchanged. See any database textbook for more information.)

Let's assume there's an object representing a database connection. Our goal will be to let the user write code like this:

```
db_connection = DatabaseConnection()
with db_connection as cursor:
    cursor.execute('insert into ...')
    cursor.execute('delete from ...')
    # ... more operations ...
```

The transaction should be committed if the code in the block runs flawlessly or rolled back if there's an exception. Here's the basic interface for `DatabaseConnection` that I'll assume:

```
class DatabaseConnection:
    # Database interface
    def cursor (self):
        "Returns a cursor object and starts a new transaction"
    def commit (self):
        "Commits current transaction"
    def rollback (self):
        "Rolls back current transaction"
```

The `__enter__()` method is pretty easy, having only to start a new transaction. For this application the resulting cursor object would be a useful result, so the method will return it. The user can then add `as cursor` to their '`with`' statement to bind the cursor to a variable name.

```
class DatabaseConnection:
    ...
    def __enter__ (self):
        # Code to start a new transaction
        cursor = self.cursor()
        return cursor
```

The `__exit__()` method is the most complicated because it's where most of the work has to be done. The method has to check if an exception occurred. If there was no exception, the transaction is committed. The transaction is rolled back if there was an exception.

In the code below, execution will just fall off the end of the function, returning the default value of `None`. `None` is false, so the exception will be re-raised automatically. If you wished, you could be more explicit and add a `return` statement at the marked location.

```
class DatabaseConnection:
    ...
    def __exit__ (self, type, value, tb):
        if tb is None:
            # No exception, so commit
            self.commit()
        else:
            # Exception occurred, so rollback.
            self.rollback()
            # return False
```

8.2 The contextlib module

The new `contextlib` module provides some functions and a decorator that are useful for writing objects for use with the '`with`' statement.

The decorator is called `contextmanager`, and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the `yield` will be executed as the `__enter__()` method, and the value yielded will be the method's return value that will get bound to the variable in the '`with`' statement's `as` clause, if any. The code after the `yield` will be executed in the `__exit__()` method. Any exception raised in the block will be raised by the `yield` statement.

Our database example from the previous section could be written using this decorator as:

```
from contextlib import contextmanager

@contextmanager
def db_transaction (connection):
    cursor = connection.cursor()
    try:
        yield cursor
    except:
        connection.rollback()
        raise
    else:
        connection.commit()

db = DatabaseConnection()
with db_transaction(db) as cursor:
    ...
```

The `contextlib` module also has a `nested(`*mgr1*`, `*mgr2*`, ...)` function that combines a number of context managers so you don't need to write nested '`with`' statements. In this example, the single '`with`' statement both starts a database transaction and acquires a thread lock:

```
lock = threading.Lock()
with nested (db_transaction(db), lock) as (cursor, locked):
    ...
```

Finally, the `closing(`*object*`)` function returns *object* so that it can be bound to a variable, and calls *object*`.close()` at the end of the block.

```
import urllib, sys
from contextlib import closing

with closing(urllib.urlopen('http://www.yahoo.com')) as f:
    for line in f:
        sys.stdout.write(line)
```

See Also:

PEP 343, "*The "with" statement*"
PEP written by Guido van Rossum and Nick Coghlan; implemented by Mike Bland, Guido van Rossum, and Neal Norwitz. The PEP shows the code generated for a '`with`' statement, which can be helpful in learning how

the statement works.

../lib/module-contextlib.html
The documentation for the `contextlib` module.

9 PEP 352: Exceptions as New-Style Classes

Exception classes can now be new-style classes, not just classic classes, and the built-in `Exception` class and all the standard built-in exceptions (`NameError`, `ValueError`, etc.) are now new-style classes.

The inheritance hierarchy for exceptions has been rearranged a bit. In 2.5, the inheritance relationships are:

```
BaseException       # New in Python 2.5
|- KeyboardInterrupt
|- SystemExit
|- Exception
   |- (all other current built-in exceptions)
```

This rearrangement was done because people often want to catch all exceptions that indicate program errors. `KeyboardInterrupt` and `SystemExit` aren't errors, though, and usually represent an explicit action such as the user hitting Control-C or code calling `sys.exit()`. A bare `except:` will catch all exceptions, so you commonly need to list `KeyboardInterrupt` and `SystemExit` in order to re-raise them. The usual pattern is:

```
try:
    ...
except (KeyboardInterrupt, SystemExit):
    raise
except:
    # Log error...
    # Continue running program...
```

In Python 2.5, you can now write `except Exception` to achieve the same result, catching all the exceptions that usually indicate errors but leaving `KeyboardInterrupt` and `SystemExit` alone. As in previous versions, a bare `except:` still catches all exceptions.

The goal for Python 3.0 is to require any class raised as an exception to derive from `BaseException` or some descendant of `BaseException`, and future releases in the Python 2.x series may begin to enforce this constraint. Therefore, I suggest you begin making all your exception classes derive from `Exception` now. It's been suggested that the bare `except:` form should be removed in Python 3.0, but Guido van Rossum hasn't decided whether to do this or not.

Raising of strings as exceptions, as in the statement `raise "Error occurred"`, is deprecated in Python 2.5 and will trigger a warning. The aim is to be able to remove the string-exception feature in a few releases.

See Also:

PEP 352, "*Required Superclass for Exceptions*"
PEP written by Brett Cannon and Guido van Rossum; implemented by Brett Cannon.

10 PEP 353: Using ssize_t as the index type

A wide-ranging change to Python's C API, using a new `Py_ssize_t` type definition instead of `int`, will permit the interpreter to handle more data on 64-bit platforms. This change doesn't affect Python's capacity on 32-bit platforms.

Various pieces of the Python interpreter used C's `int` type to store sizes or counts; for example, the number of items in a list or tuple were stored in an `int`. The C compilers for most 64-bit platforms still define `int` as a 32-bit type, so that meant that lists could only hold up to `2**31 - 1` = 2147483647 items. (There are actually a few different programming models that 64-bit C compilers can use – see http://www.unix.org/version2/whatsnew/lp64_wp.html for a discussion – but the most commonly available model leaves `int` as 32 bits.)

A limit of 2147483647 items doesn't really matter on a 32-bit platform because you'll run out of memory before hitting the length limit. Each list item requires space for a pointer, which is 4 bytes, plus space for a `PyObject` representing the item. 2147483647*4 is already more bytes than a 32-bit address space can contain.

It's possible to address that much memory on a 64-bit platform, however. The pointers for a list that size would only require 16 GiB of space, so it's not unreasonable that Python programmers might construct lists that large. Therefore, the Python interpreter had to be changed to use some type other than `int`, and this will be a 64-bit type on 64-bit platforms. The change will cause incompatibilities on 64-bit machines, so it was deemed worth making the transition now, while the number of 64-bit users is still relatively small. (In 5 or 10 years, we may *all* be on 64-bit machines, and the transition would be more painful then.)

This change most strongly affects authors of C extension modules. Python strings and container types such as lists and tuples now use `Py_ssize_t` to store their size. Functions such as `PyList_Size()` now return `Py_ssize_t`. Code in extension modules may therefore need to have some variables changed to `Py_ssize_t`.

The `PyArg_ParseTuple()` and `Py_BuildValue()` functions have a new conversion code, '`n`', for `Py_ssize_t`. `PyArg_ParseTuple()`'s '`s#`' and '`t#`' still output `int` by default, but you can define the macro `PY_SSIZE_T_CLEAN` before including 'Python.h' to make them return `Py_ssize_t`.

PEP 353 has a section on conversion guidelines that extension authors should read to learn about supporting 64-bit platforms.

See Also:

PEP 353, "*Using ssize_t as the index type*"
PEP written and implemented by Martin von Löwis.

11 PEP 357: The '__index__' method

The NumPy developers had a problem that could only be solved by adding a new special method, `__index__`. When using slice notation, as in `[`*start*`:`*stop*`:`*step*`]`, the values of the *start*, *stop*, and *step* indexes must all be either integers or long integers. NumPy defines a variety of specialized integer types corresponding to unsigned and signed integers of 8, 16, 32, and 64 bits, but there was no way to signal that these types could be used as slice indexes.

Slicing can't just use the existing `__int__` method because that method is also used to implement coercion to integers. If slicing used `__int__`, floating-point numbers would also become legal slice indexes and that's clearly an undesirable behaviour.

Instead, a new special method called `__index__` was added. It takes no arguments and returns an integer giving the slice index to use. For example:

```
class C:
    def __index__ (self):
        return self.value
```

The return value must be either a Python integer or long integer. The interpreter will check that the type returned is correct, and raises a `TypeError` if this requirement isn't met.

A corresponding `nb_index` slot was added to the C-level `PyNumberMethods` structure to let C extensions implement this protocol. `PyNumber_Index(`*obj*`)` can be used in extension code to call the `__index__` function and retrieve its result.

See Also:

PEP 357, "*Allowing Any Object to be Used for Slicing*"
PEP written and implemented by Travis Oliphant.

12 Other Language Changes

Here are all of the changes that Python 2.5 makes to the core Python language.

- The `dict` type has a new hook for letting subclasses provide a default value when a key isn't contained in the dictionary. When a key isn't found, the dictionary's `__missing__(`*key*`)` method will be called. This hook is used to implement the new `defaultdict` class in the `collections` module. The following example defines a dictionary that returns zero for any missing key:

```
class zerodict (dict):
    def __missing__ (self, key):
        return 0

d = zerodict({1:1, 2:2})
print d[1], d[2]   # Prints 1, 2
print d[3], d[4]   # Prints 0, 0
```

- Both 8-bit and Unicode strings have new `partition(sep)` and `rpartition(sep)` methods that simplify a common use case.

  The `find(S)` method is often used to get an index which is then used to slice the string and obtain the pieces that are before and after the separator. `partition(sep)` condenses this pattern into a single method call that returns a 3-tuple containing the substring before the separator, the separator itself, and the substring after the separator. If the separator isn't found, the first element of the tuple is the entire string and the other two elements are empty. `rpartition(sep)` also returns a 3-tuple but starts searching from the end of the string; the '`r`' stands for 'reverse'.

  Some examples:

```
>>> ('http://www.python.org').partition('://')
('http', '://', 'www.python.org')
>>> ('file:/usr/share/doc/index.html').partition('://')
('file:/usr/share/doc/index.html', '', '')
>>> (u'Subject: a quick question').partition(':')
(u'Subject', u':', u' a quick question')
>>> 'www.python.org'.rpartition('.')
('www.python', '.', 'org')
>>> 'www.python.org'.rpartition(':')
('', '', 'www.python.org')
```

  (Implemented by Fredrik Lundh following a suggestion by Raymond Hettinger.)

- The `startswith()` and `endswith()` methods of string types now accept tuples of strings to check for.

```
def is_image_file (filename):
    return filename.endswith(('.gif', '.jpg', '.tiff'))
```

(Implemented by Georg Brandl following a suggestion by Tom Lynn.)

- The `min()` and `max()` built-in functions gained a `key` keyword parameter analogous to the `key` argument for `sort()`. This parameter supplies a function that takes a single argument and is called for every value in the list; `min()`/`max()` will return the element with the smallest/largest return value from this function. For example, to find the longest string in a list, you can do:

```
L = ['medium', 'longest', 'short']
# Prints 'longest'
print max(L, key=len)
# Prints 'short', because lexicographically 'short' has the largest value
print max(L)
```

  (Contributed by Steven Bethard and Raymond Hettinger.)

- Two new built-in functions, `any()` and `all()`, evaluate whether an iterator contains any true or false values. `any()` returns `True` if any value returned by the iterator is true; otherwise it will return `False`. `all()` returns `True` only if all of the values returned by the iterator evaluate as true. (Suggested by Guido van Rossum, and implemented by Raymond Hettinger.)

- The result of a class's `__hash__()` method can now be either a long integer or a regular integer. If a long integer is returned, the hash of that value is taken. In earlier versions the hash value was required to be a regular integer, but in 2.5 the `id()` built-in was changed to always return non-negative numbers, and users often seem to use `id(self)` in `__hash__()` methods (though this is discouraged).

- ASCII is now the default encoding for modules. It's now a syntax error if a module contains string literals with 8-bit characters but doesn't have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error. See PEP 263 for how to declare a module's encoding; for example, you might add a line like this near the top of the source file:

```
# -*- coding: latin1 -*-
```

- A new warning, `UnicodeWarning`, is triggered when you attempt to compare a Unicode string and an 8-bit string that can't be converted to Unicode using the default ASCII encoding. The result of the comparison is false:

```
>>> chr(128) == unichr(128)   # Can't convert chr(128) to Unicode
__main__:1: UnicodeWarning: Unicode equal comparison failed
  to convert both arguments to Unicode - interpreting them
  as being unequal
False
>>> chr(127) == unichr(127)   # chr(127) can be converted
True
```

  Previously this would raise a `UnicodeDecodeError` exception, but in 2.5 this could result in puzzling problems when accessing a dictionary. If you looked up `unichr(128)` and `chr(128)` was being used as a key, you'd get a `UnicodeDecodeError` exception. Other changes in 2.5 resulted in this exception being raised instead of suppressed by the code in 'dictobject.c' that implements dictionaries.

  Raising an exception for such a comparison is strictly correct, but the change might have broken code, so instead `UnicodeWarning` was introduced.

  (Implemented by Marc-André Lemburg.)

- One error that Python programmers sometimes make is forgetting to include an '__init__.py' module in a package directory. Debugging this mistake can be confusing, and usually requires running Python with the **-v** switch to log all the paths searched. In Python 2.5, a new `ImportWarning` warning is triggered when an import would have picked up a directory as a package but no '__init__.py' was found. This warning is silently ignored by default; provide the **-Wd** option when running the Python executable to display the warning message. (Implemented by Thomas Wouters.)

- The list of base classes in a class definition can now be empty. As an example, this is now legal:

```
class C():
    pass
```

(Implemented by Brett Cannon.)

12.1 Interactive Interpreter Changes

In the interactive interpreter, `quit` and `exit` have long been strings so that new users get a somewhat helpful message when they try to quit:

```
>>> quit
'Use Ctrl-D (i.e. EOF) to exit.'
```

In Python 2.5, `quit` and `exit` are now objects that still produce string representations of themselves, but are also callable. Newbies who try `quit()` or `exit()` will now exit the interpreter as they expect. (Implemented by Georg Brandl.)

The Python executable now accepts the standard long options **--help** and **--version**; on Windows, it also accepts the **/?** option for displaying a help message. (Implemented by Georg Brandl.)

12.2 Optimizations

Several of the optimizations were developed at the NeedForSpeed sprint, an event held in Reykjavik, Iceland, from May 21–28 2006. The sprint focused on speed enhancements to the CPython implementation and was funded by EWT LLC with local support from CCP Games. Those optimizations added at this sprint are specially marked in the following list.

- When they were introduced in Python 2.4, the built-in `set` and `frozenset` types were built on top of Python's dictionary type. In 2.5 the internal data structure has been customized for implementing sets, and as a result sets will use a third less memory and are somewhat faster. (Implemented by Raymond Hettinger.)
- The speed of some Unicode operations, such as finding substrings, string splitting, and character map encoding and decoding, has been improved. (Substring search and splitting improvements were added by Fredrik Lundh and Andrew Dalke at the NeedForSpeed sprint. Character maps were improved by Walter Dörwald and Martin von Löwis.)
- The `long(`*str*`, `*base*`)` function is now faster on long digit strings because fewer intermediate results are calculated. The peak is for strings of around 800–1000 digits where the function is 6 times faster. (Contributed by Alan McIntyre and committed at the NeedForSpeed sprint.)
- It's now illegal to mix iterating over a file with `for line in` *file* and calling the file object's `read()/readline()/readlines()` methods. Iteration uses an internal buffer and the `read*()` methods don't use that buffer. Instead they would return the data following the buffer, causing the data to appear out of order. Mixing iteration and these methods will now trigger a `ValueError` from the `read*()` method. (Implemented by Thomas Wouters.)

- The `struct` module now compiles structure format strings into an internal representation and caches this representation, yielding a 20% speedup. (Contributed by Bob Ippolito at the NeedForSpeed sprint.)
- The `re` module got a 1 or 2% speedup by switching to Python's allocator functions instead of the system's `malloc()` and `free()`. (Contributed by Jack Diederich at the NeedForSpeed sprint.)
- The code generator's peephole optimizer now performs simple constant folding in expressions. If you write something like `a = 2+3`, the code generator will do the arithmetic and produce code corresponding to `a = 5`. (Proposed and implemented by Raymond Hettinger.)
- Function calls are now faster because code objects now keep the most recently finished frame (a "zombie frame") in an internal field of the code object, reusing it the next time the code object is invoked. (Original patch by Michael Hudson, modified by Armin Rigo and Richard Jones; committed at the NeedForSpeed sprint.)

  Frame objects are also slightly smaller, which may improve cache locality and reduce memory usage a bit. (Contributed by Neal Norwitz.)
- Python's built-in exceptions are now new-style classes, a change that speeds up instantiation considerably. Exception handling in Python 2.5 is therefore about 30% faster than in 2.4. (Contributed by Richard Jones, Georg Brandl and Sean Reifschneider at the NeedForSpeed sprint.)
- Importing now caches the paths tried, recording whether they exist or not so that the interpreter makes fewer `open()` and `stat()` calls on startup. (Contributed by Martin von Löwis and Georg Brandl.)

13 New, Improved, and Removed Modules

The standard library received many enhancements and bug fixes in Python 2.5. 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 SVN logs for all the details.

- The `audioop` module now supports the a-LAW encoding, and the code for u-LAW encoding has been improved. (Contributed by Lars Immisch.)
- The `codecs` module gained support for incremental codecs. The `codec.lookup()` function now returns a `CodecInfo` instance instead of a tuple. `CodecInfo` instances behave like a 4-tuple to preserve backward compatibility but also have the attributes `encode`, `decode`, `incrementalencoder`, `incrementaldecoder`, `streamwriter`, and `streamreader`. Incremental codecs can receive input and produce output in multiple chunks; the output is the same as if the entire input was fed to the non-incremental codec. See the `codecs` module documentation for details. (Designed and implemented by Walter Dörwald.)
- The `collections` module gained a new type, `defaultdict`, that subclasses the standard `dict` type. The new type mostly behaves like a dictionary but constructs a default value when a key isn't present, automatically adding it to the dictionary for the requested key value.

  The first argument to `defaultdict`'s constructor is a factory function that gets called whenever a key is requested but not found. This factory function receives no arguments, so you can use built-in type constructors such as `list()` or `int()`. For example, you can make an index of words based on their initial letter like this:

  ```
  words = """Nel mezzo del cammin di nostra vita
  mi ritrovai per una selva oscura
  che la diritta via era smarrita""".lower().split()

  index = defaultdict(list)

  for w in words:
      init_letter = w[0]
      index[init_letter].append(w)
  ```

Printing `index` results in the following output:

```
defaultdict(<type 'list'>, {'c': ['cammin', 'che'], 'e': ['era'],
        'd': ['del', 'di', 'diritta'], 'm': ['mezzo', 'mi'],
        'l': ['la'], 'o': ['oscura'], 'n': ['nel', 'nostra'],
        'p': ['per'], 's': ['selva', 'smarrita'],
        'r': ['ritrovai'], 'u': ['una'], 'v': ['vita', 'via']}
```

(Contributed by Guido van Rossum.)

- The `deque` double-ended queue type supplied by the `collections` module now has a `remove(`*value*`)` method that removes the first occurrence of *value* in the queue, raising `ValueError` if the value isn't found. (Contributed by Raymond Hettinger.)
- New module: The `contextlib` module contains helper functions for use with the new '`with`' statement. See section 8.2 for more about this module.
- New module: The `cProfile` module is a C implementation of the existing `profile` module that has much lower overhead. The module's interface is the same as `profile`: you run `cProfile.run('main()')` to profile a function, can save profile data to a file, etc. It's not yet known if the Hotshot profiler, which is also written in C but doesn't match the `profile` module's interface, will continue to be maintained in future versions of Python. (Contributed by Armin Rigo.)

  Also, the `pstats` module for analyzing the data measured by the profiler now supports directing the output to any file object by supplying a *stream* argument to the `Stats` constructor. (Contributed by Skip Montanaro.)
- The `csv` module, which parses files in comma-separated value format, received several enhancements and a number of bugfixes. You can now set the maximum size in bytes of a field by calling the `csv.field_size_limit(`*new_limit*`)` function; omitting the *new_limit* argument will return the currently-set limit. The `reader` class now has a `line_num` attribute that counts the number of physical lines read from the source; records can span multiple physical lines, so `line_num` is not the same as the number of records read.

  The CSV parser is now stricter about multi-line quoted fields. Previously, if a line ended within a quoted field without a terminating newline character, a newline would be inserted into the returned field. This behavior caused problems when reading files that contained carriage return characters within fields, so the code was changed to return the field without inserting newlines. As a consequence, if newlines embedded within fields are important, the input should be split into lines in a manner that preserves the newline characters.

  (Contributed by Skip Montanaro and Andrew McNamara.)
- The `datetime` class in the `datetime` module now has a `strptime(`*string*`, `*format*`)` method for parsing date strings, contributed by Josh Spoerri. It uses the same format characters as `time.strptime()` and `time.strftime()`:

  ```
  from datetime import datetime

  ts = datetime.strptime('10:13:15 2006-03-07',
                         '%H:%M:%S %Y-%m-%d')
  ```
- The `SequenceMatcher.get_matching_blocks()` method in the `difflib` module now guarantees to return a minimal list of blocks describing matching subsequences. Previously, the algorithm would occasionally break a block of matching elements into two list entries. (Enhancement by Tim Peters.)
- The `doctest` module gained a `SKIP` option that keeps an example from being executed at all. This is intended for code snippets that are usage examples intended for the reader and aren't actually test cases.

An *encoding* parameter was added to the `testfile()` function and the `DocFileSuite` class to specify the file's encoding. This makes it easier to use non-ASCII characters in tests contained within a docstring. (Contributed by Bjorn Tillenius.)

- The `email` package has been updated to version 4.0. (Contributed by Barry Warsaw.)

- The `fileinput` module was made more flexible. Unicode filenames are now supported, and a *mode* parameter that defaults to `"r"` was added to the `input()` function to allow opening files in binary or universal-newline mode. Another new parameter, *openhook*, lets you use a function other than `open()` to open the input files. Once you're iterating over the set of files, the `FileInput` object's new `fileno()` returns the file descriptor for the currently opened file. (Contributed by Georg Brandl.)

- In the `gc` module, the new `get_count()` function returns a 3-tuple containing the current collection counts for the three GC generations. This is accounting information for the garbage collector; when these counts reach a specified threshold, a garbage collection sweep will be made. The existing `gc.collect()` function now takes an optional *generation* argument of 0, 1, or 2 to specify which generation to collect. (Contributed by Barry Warsaw.)

- The `nsmallest()` and `nlargest()` functions in the `heapq` module now support a `key` keyword parameter similar to the one provided by the `min()`/`max()` functions and the `sort()` methods. For example:

```
>>> import heapq
>>> L = ["short", 'medium', 'longest', 'longer still']
>>> heapq.nsmallest(2, L)  # Return two lowest elements, lexicographically
['longer still', 'longest']
>>> heapq.nsmallest(2, L, key=len)   # Return two shortest elements
['short', 'medium']
```

  (Contributed by Raymond Hettinger.)

- The `itertools.islice()` function now accepts `None` for the start and step arguments. This makes it more compatible with the attributes of slice objects, so that you can now write the following:

```
s = slice(5)     # Create slice object
itertools.islice(iterable, s.start, s.stop, s.step)
```

  (Contributed by Raymond Hettinger.)

- The `format()` function in the `locale` module has been modified and two new functions were added, `format_string()` and `currency()`.

  The `format()` function's *val* parameter could previously be a string as long as no more than one %char specifier appeared; now the parameter must be exactly one %char specifier with no surrounding text. An optional *monetary* parameter was also added which, if `True`, will use the locale's rules for formatting currency in placing a separator between groups of three digits.

  To format strings with multiple %char specifiers, use the new `format_string()` function that works like `format()` but also supports mixing %char specifiers with arbitrary text.

  A new `currency()` function was also added that formats a number according to the current locale's settings.

  (Contributed by Georg Brandl.)

- The `mailbox` module underwent a massive rewrite to add the capability to modify mailboxes in addition to reading them. A new set of classes that include `mbox`, `MH`, and `Maildir` are used to read mailboxes, and have an `add(`*message*`)` method to add messages, `remove(`*key*`)` to remove messages, and `lock()`/`unlock()` to lock/unlock the mailbox. The following example converts a maildir-format mailbox into an mbox-format one:

```
import mailbox

# 'factory=None' uses email.Message.Message as the class representing
# individual messages.
src = mailbox.Maildir('maildir', factory=None)
dest = mailbox.mbox('/tmp/mbox')

for msg in src:
    dest.add(msg)
```

(Contributed by Gregory K. Johnson. Funding was provided by Google's 2005 Summer of Code.)

- New module: the `msilib` module allows creating Microsoft Installer '.msi' files and CAB files. Some support for reading the '.msi' database is also included. (Contributed by Martin von Löwis.)

- The `nis` module now supports accessing domains other than the system default domain by supplying a *domain* argument to the `nis.match()` and `nis.maps()` functions. (Contributed by Ben Bell.)

- The `operator` module's `itemgetter()` and `attrgetter()` functions now support multiple fields. A call such as `operator.attrgetter('a', 'b')` will return a function that retrieves the `a` and `b` attributes. Combining this new feature with the `sort()` method's `key` parameter lets you easily sort lists using multiple fields. (Contributed by Raymond Hettinger.)

- The `optparse` module was updated to version 1.5.1 of the Optik library. The `OptionParser` class gained an `epilog` attribute, a string that will be printed after the help message, and a `destroy()` method to break reference cycles created by the object. (Contributed by Greg Ward.)

- The `os` module underwent several changes. The `stat_float_times` variable now defaults to true, meaning that `os.stat()` will now return time values as floats. (This doesn't necessarily mean that `os.stat()` will return times that are precise to fractions of a second; not all systems support such precision.)

  Constants named `os.SEEK_SET`, `os.SEEK_CUR`, and `os.SEEK_END` have been added; these are the parameters to the `os.lseek()` function. Two new constants for locking are `os.O_SHLOCK` and `os.O_EXLOCK`.

  Two new functions, `wait3()` and `wait4()`, were added. They're similar the `waitpid()` function which waits for a child process to exit and returns a tuple of the process ID and its exit status, but `wait3()` and `wait4()` return additional information. `wait3()` doesn't take a process ID as input, so it waits for any child process to exit and returns a 3-tuple of *process-id*, *exit-status*, *resource-usage* as returned from the `resource.getrusage()` function. `wait4(`*pid*`)` does take a process ID. (Contributed by Chad J. Schroeder.)

  On FreeBSD, the `os.stat()` function now returns times with nanosecond resolution, and the returned object now has `st_gen` and `st_birthtime`. The `st_flags` member is also available, if the platform supports it. (Contributed by Antti Louko and Diego Pettenò.)

- The Python debugger provided by the `pdb` module can now store lists of commands to execute when a breakpoint is reached and execution stops. Once breakpoint #1 has been created, enter '`commands 1`' and enter a series of commands to be executed, finishing the list with '`end`'. The command list can include commands that resume execution, such as '`continue`' or '`next`'. (Contributed by Grégoire Dooms.)

- The `pickle` and `cPickle` modules no longer accept a return value of `None` from the `__reduce__()` method; the method must return a tuple of arguments instead. The ability to return `None` was deprecated in Python 2.4, so this completes the removal of the feature.

- The `pkgutil` module, containing various utility functions for finding packages, was enhanced to support PEP 302's import hooks and now also works for packages stored in ZIP-format archives. (Contributed by Phillip J. Eby.)

- The pybench benchmark suite by Marc-André Lemburg is now included in the 'Tools/pybench' directory. The pybench suite is an improvement on the commonly used 'pystone.py' program because pybench provides a more detailed measurement of the interpreter's speed. It times particular operations such as function calls, tuple slicing, method lookups, and numeric operations, instead of performing many different operations and reducing the result to a single number as 'pystone.py' does.
- The `pyexpat` module now uses version 2.0 of the Expat parser. (Contributed by Trent Mick.)
- The `Queue` class provided by the `Queue` module gained two new methods. `join()` blocks until all items in the queue have been retrieved and all processing work on the items have been completed. Worker threads call the other new method, `task_done()`, to signal that processing for an item has been completed. (Contributed by Raymond Hettinger.)
- The old `regex` and `regsub` modules, which have been deprecated ever since Python 2.0, have finally been deleted. Other deleted modules: `statcache`, `tzparse`, `whrandom`.
- Also deleted: the 'lib-old' directory, which includes ancient modules such as `dircmp` and `ni`, was removed. 'lib-old' wasn't on the default `sys.path`, so unless your programs explicitly added the directory to `sys.path`, this removal shouldn't affect your code.
- The `rlcompleter` module is no longer dependent on importing the `readline` module and therefore now works on non-UNIX platforms. (Patch from Robert Kiendl.)
- The `SimpleXMLRPCServer` and `DocXMLRPCServer` classes now have a `rpc_paths` attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only `'/'` and `'/RPC2'`. Setting `rpc_paths` to `None` or an empty tuple disables this path checking.
- The `socket` module now supports `AF_NETLINK` sockets on Linux, thanks to a patch from Philippe Biondi. Netlink sockets are a Linux-specific mechanism for communications between a user-space process and kernel code; an introductory article about them is at http://www.linuxjournal.com/article/7356. In Python code, netlink addresses are represented as a tuple of 2 integers, `(`*pid*`,` *group_mask*`)`.

  Two new methods on socket objects, `recv_into(`*buffer*`)` and `recvfrom_into(`*buffer*`)`, store the received data in an object that supports the buffer protocol instead of returning the data as a string. This means you can put the data directly into an array or a memory-mapped file.

  Socket objects also gained `getfamily()`, `gettype()`, and `getproto()` accessor methods to retrieve the family, type, and protocol values for the socket.
- New module: the `spwd` module provides functions for accessing the shadow password database on systems that support shadow passwords.
- The `struct` is now faster because it compiles format strings into `Struct` objects with `pack()` and `unpack()` methods. This is similar to how the `re` module lets you create compiled regular expression objects. You can still use the module-level `pack()` and `unpack()` functions; they'll create `Struct` objects and cache them. Or you can use `Struct` instances directly:

```
s = struct.Struct('ih3s')

data = s.pack(1972, 187, 'abc')
year, number, name = s.unpack(data)
```

  You can also pack and unpack data to and from buffer objects directly using the `pack_into(`*buffer*`,` *offset*`,` *v1*`,` *v2*`, ...)` and `unpack_from(`*buffer*`,` *offset*`)` methods. This lets you store data directly into an array or a memory-mapped file.

  (`Struct` objects were implemented by Bob Ippolito at the NeedForSpeed sprint. Support for buffer objects was added by Martin Blais, also at the NeedForSpeed sprint.)

- The Python developers switched from CVS to Subversion during the 2.5 development process. Information about the exact build version is available as the `sys.subversion` variable, a 3-tuple of `(`*interpreter-name*`,` *branch-name*`,` *revision-range*`)`. For example, at the time of writing my copy of 2.5 was reporting `('CPython', 'trunk', '45313:45315')`.

  This information is also available to C extensions via the `Py_GetBuildInfo()` function that returns a string of build information like this: `"trunk:45355:45356M, Apr 13 2006, 07:42:19"`. (Contributed by Barry Warsaw.)

- Another new function, `sys._current_frames()`, returns the current stack frames for all running threads as a dictionary mapping thread identifiers to the topmost stack frame currently active in that thread at the time the function is called. (Contributed by Tim Peters.)

- The `TarFile` class in the `tarfile` module now has an `extractall()` method that extracts all members from the archive into the current working directory. It's also possible to set a different directory as the extraction target, and to unpack only a subset of the archive's members.

  The compression used for a tarfile opened in stream mode can now be autodetected using the mode `'r|*'`. (Contributed by Lars Gustäbel.)

- The `threading` module now lets you set the stack size used when new threads are created. The `stack_size(`[*size*]`)` function returns the currently configured stack size, and supplying the optional *size* parameter sets a new value. Not all platforms support changing the stack size, but Windows, POSIX threading, and OS/2 all do. (Contributed by Andrew MacIntyre.)

- The `unicodedata` module has been updated to use version 4.1.0 of the Unicode character database. Version 3.2.0 is required by some specifications, so it's still available as `unicodedata.ucd_3_2_0`.

- New module: the `uuid` module generates universally unique identifiers (UUIDs) according to RFC 4122. The RFC defines several different UUID versions that are generated from a starting string, from system properties, or purely randomly. This module contains a `UUID` class and functions named `uuid1()`, `uuid3()`, `uuid4()`, and `uuid5()` to generate different versions of UUID. (Version 2 UUIDs are not specified in RFC 4122 and are not supported by this module.)

  ```
  >>> import uuid
  >>> # make a UUID based on the host ID and current time
  >>> uuid.uuid1()
  UUID('a8098c1a-f86e-11da-bd1a-00112444be1e')

  >>> # make a UUID using an MD5 hash of a namespace UUID and a name
  >>> uuid.uuid3(uuid.NAMESPACE_DNS, 'python.org')
  UUID('6fa459ea-ee8a-3ca4-894e-db77e160355e')

  >>> # make a random UUID
  >>> uuid.uuid4()
  UUID('16fd2706-8baf-433b-82eb-8c7fada847da')

  >>> # make a UUID using a SHA-1 hash of a namespace UUID and a name
  >>> uuid.uuid5(uuid.NAMESPACE_DNS, 'python.org')
  UUID('886313e1-3b8a-5372-9b90-0c9aee199e5d')
  ```

  (Contributed by Ka-Ping Yee.)

- The `weakref` module's `WeakKeyDictionary` and `WeakValueDictionary` types gained new methods for iterating over the weak references contained in the dictionary. `iterkeyrefs()` and `keyrefs()` methods were added to `WeakKeyDictionary`, and `itervaluerefs()` and `valuerefs()` were added to `WeakValueDictionary`. (Contributed by Fred L. Drake, Jr.)

- The webbrowser module received a number of enhancements. It's now usable as a script with python -m webbrowser, taking a URL as the argument; there are a number of switches to control the behaviour (**-n** for a new browser window, **-t** for a new tab). New module-level functions, open_new() and open_new_tab(), were added to support this. The module's open() function supports an additional feature, an *autoraise* parameter that signals whether to raise the open window when possible. A number of additional browsers were added to the supported list such as Firefox, Opera, Konqueror, and elinks. (Contributed by Oleg Broytmann and Georg Brandl.)
- The xmlrpclib module now supports returning datetime objects for the XML-RPC date type. Supply use_datetime=True to the loads() function or the Unmarshaller class to enable this feature. (Contributed by Skip Montanaro.)
- The zipfile module now supports the ZIP64 version of the format, meaning that a .zip archive can now be larger than 4 GiB and can contain individual files larger than 4 GiB. (Contributed by Ronald Oussoren.)
- The zlib module's Compress and Decompress objects now support a copy() method that makes a copy of the object's internal state and returns a new Compress or Decompress object. (Contributed by Chris AtLee.)

13.1 The ctypes package

The ctypes package, written by Thomas Heller, has been added to the standard library. ctypes lets you call arbitrary functions in shared libraries or DLLs. Long-time users may remember the dl module, which provides functions for loading shared libraries and calling functions in them. The ctypes package is much fancier.

To load a shared library or DLL, you must create an instance of the CDLL class and provide the name or path of the shared library or DLL. Once that's done, you can call arbitrary functions by accessing them as attributes of the CDLL object.

```
import ctypes

libc = ctypes.CDLL('libc.so.6')
result = libc.printf("Line of output\n")
```

Type constructors for the various C types are provided: c_int, c_float, c_double, c_char_p (equivalent to char *), and so forth. Unlike Python's types, the C versions are all mutable; you can assign to their value attribute to change the wrapped value. Python integers and strings will be automatically converted to the corresponding C types, but for other types you must call the correct type constructor. (And I mean *must*; getting it wrong will often result in the interpreter crashing with a segmentation fault.)

You shouldn't use c_char_p with a Python string when the C function will be modifying the memory area, because Python strings are supposed to be immutable; breaking this rule will cause puzzling bugs. When you need a modifiable memory area, use create_string_buffer():

```
s = "this is a string"
buf = ctypes.create_string_buffer(s)
libc.strfry(buf)
```

C functions are assumed to return integers, but you can set the restype attribute of the function object to change this:

```
>>> libc.atof('2.71828')
-1783957616
>>> libc.atof.restype = ctypes.c_double
>>> libc.atof('2.71828')
2.71828
```

`ctypes` also provides a wrapper for Python's C API as the `ctypes.pythonapi` object. This object does *not* release the global interpreter lock before calling a function, because the lock must be held when calling into the interpreter's code. There's a `py_object()` type constructor that will create a `PyObject *` pointer. A simple usage:

```
import ctypes

d = {}
ctypes.pythonapi.PyObject_SetItem(ctypes.py_object(d),
          ctypes.py_object("abc"),  ctypes.py_object(1))
# d is now {'abc', 1}.
```

Don't forget to use `py_object()`; if it's omitted you end up with a segmentation fault.

`ctypes` has been around for a while, but people still write and distribution hand-coded extension modules because you can't rely on `ctypes` being present. Perhaps developers will begin to write Python wrappers atop a library accessed through `ctypes` instead of extension modules, now that `ctypes` is included with core Python.

See Also:

http://starship.python.net/crew/theller/ctypes/
The ctypes web page, with a tutorial, reference, and FAQ.

../lib/module-ctypes.html
The documentation for the `ctypes` module.

13.2 The ElementTree package

A subset of Fredrik Lundh's ElementTree library for processing XML has been added to the standard library as `xml.etree`. The available modules are `ElementTree`, `ElementPath`, and `ElementInclude` from ElementTree 1.2.6. The `cElementTree` accelerator module is also included.

The rest of this section will provide a brief overview of using ElementTree. Full documentation for ElementTree is available at http://effbot.org/zone/element-index.htm.

ElementTree represents an XML document as a tree of element nodes. The text content of the document is stored as the `.text` and `.tail` attributes of (This is one of the major differences between ElementTree and the Document Object Model; in the DOM there are many different types of node, including `TextNode`.)

The most commonly used parsing function is `parse()`, that takes either a string (assumed to contain a filename) or a file-like object and returns an `ElementTree` instance:

```
from xml.etree import ElementTree as ET

tree = ET.parse('ex-1.xml')

feed = urllib.urlopen(
          'http://planet.python.org/rss10.xml')
tree = ET.parse(feed)
```

Once you have an `ElementTree` instance, you can call its `getroot()` method to get the root `Element` node.

There's also an `XML()` function that takes a string literal and returns an `Element` node (not an `ElementTree`). This function provides a tidy way to incorporate XML fragments, approaching the convenience of an XML literal:

```
svg = ET.XML("""<svg width="10px" version="1.0">
             </svg>""")
svg.set('height', '320px')
svg.append(elem1)
```

Each XML element supports some dictionary-like and some list-like access methods. Dictionary-like operations are used to access attribute values, and list-like operations are used to access child nodes.

Operation	**Result**
`elem[n]`	Returns n'th child element.
`elem[m:n]`	Returns list of m'th through n'th child elements.
`len(elem)`	Returns number of child elements.
`list(elem)`	Returns list of child elements.
`elem.append(elem2)`	Adds *elem2* as a child.
`elem.insert(index, elem2)`	Inserts *elem2* at the specified location.
`del elem[n]`	Deletes n'th child element.
`elem.keys()`	Returns list of attribute names.
`elem.get(name)`	Returns value of attribute *name*.
`elem.set(name, value)`	Sets new value for attribute *name*.
`elem.attrib`	Retrieves the dictionary containing attributes.
`del elem.attrib[name]`	Deletes attribute *name*.

Comments and processing instructions are also represented as `Element` nodes. To check if a node is a comment or processing instructions:

```
if elem.tag is ET.Comment:
    ...
elif elem.tag is ET.ProcessingInstruction:
    ...
```

To generate XML output, you should call the `ElementTree.write()` method. Like `parse()`, it can take either a string or a file-like object:

```
# Encoding is US-ASCII
tree.write('output.xml')

# Encoding is UTF-8
f = open('output.xml', 'w')
tree.write(f, encoding='utf-8')
```

(Caution: the default encoding used for output is ASCII. For general XML work, where an element's name may contain arbitrary Unicode characters, ASCII isn't a very useful encoding because it will raise an exception if an element's name contains any characters with values greater than 127. Therefore, it's best to specify a different encoding such as UTF-8 that can handle any Unicode character.)

This section is only a partial description of the ElementTree interfaces. Please read the package's official documentation for more details.

See Also:

http://effbot.org/zone/element-index.htm
Official documentation for ElementTree.

13.3 The hashlib package

A new `hashlib` module, written by Gregory P. Smith, has been added to replace the `md5` and `sha` modules. `hashlib` adds support for additional secure hashes (SHA-224, SHA-256, SHA-384, and SHA-512). When available, the module uses OpenSSL for fast platform optimized implementations of algorithms.

The old `md5` and `sha` modules still exist as wrappers around hashlib to preserve backwards compatibility. The new module's interface is very close to that of the old modules, but not identical. The most significant difference is that the constructor functions for creating new hashing objects are named differently.

```
# Old versions
h = md5.md5()
h = md5.new()

# New version
h = hashlib.md5()

# Old versions
h = sha.sha()
h = sha.new()

# New version
h = hashlib.sha1()

# Hash that weren't previously available
h = hashlib.sha224()
h = hashlib.sha256()
h = hashlib.sha384()
h = hashlib.sha512()

# Alternative form
h = hashlib.new('md5')          # Provide algorithm as a string
```

Once a hash object has been created, its methods are the same as before: `update(`*string*`)` hashes the specified string into the current digest state, `digest()` and `hexdigest()` return the digest value as a binary string or a string of hex digits, and `copy()` returns a new hashing object with the same digest state.

See Also:

../lib/module-hashlib.html
  The documentation for the `hashlib` module.

13.4 The sqlite3 package

The pysqlite module (http://www.pysqlite.org), a wrapper for the SQLite embedded database, has been added to the standard library under the package name `sqlite3`.

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

pysqlite was written by Gerhard Häring and provides a SQL interface compliant with the DB-API 2.0 specification described by PEP 249.

If you're compiling the Python source yourself, note that the source tree doesn't include the SQLite code, only the wrapper module. You'll need to have the SQLite libraries and headers installed before compiling Python, and the build process will compile the module when the necessary headers are available.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the '/tmp/example' file:

```
conn = sqlite3.connect('/tmp/example')
```

You can also supply the special name '`:memory:`' to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```
c = conn.cursor()

# Create table
c.execute('''create table stocks
(date text, trans text, symbol text,
 qty real, price real)''')

# Insert a row of data
c.execute("""insert into stocks
          values ('2006-01-05','BUY','RHAT',100,35.14)""")
```

Usually your SQL operations will need to use values from Python variables. You shouldn't assemble your query using Python's string operations because doing so is insecure; it makes your program vulnerable to an SQL injection attack.

Instead, use the DB-API's parameter substitution. Put '`?`' as a placeholder wherever you want to use a value, and then provide a tuple of values as the second argument to the cursor's `execute()` method. (Other database modules may use a different placeholder, such as '`%s`' or '`:1`'.) For example:

```
# Never do this -- insecure!
symbol = 'IBM'
c.execute("... where symbol = '%s'" % symbol)

# Do this instead
t = (symbol,)
c.execute('select * from stocks where symbol=?', t)

# Larger example
for t in (('2006-03-28', 'BUY', 'IBM', 1000, 45.00),
          ('2006-04-05', 'BUY', 'MSOFT', 1000, 72.00),
          ('2006-04-06', 'SELL', 'IBM', 500, 53.00),
         ):
    c.execute('insert into stocks values (?,?,?,?,?)', t)
```

To retrieve data after executing a SELECT statement, you can either treat the cursor as an iterator, call the cursor's `fetchone()` method to retrieve a single matching row, or call `fetchall()` to get a list of the matching rows.

This example uses the iterator form:

```
>>> c = conn.cursor()
>>> c.execute('select * from stocks order by price')
>>> for row in c:
...    print row
...
(u'2006-01-05', u'BUY', u'RHAT', 100, 35.140000000000001)
(u'2006-03-28', u'BUY', u'IBM', 1000, 45.0)
(u'2006-04-06', u'SELL', u'IBM', 500, 53.0)
(u'2006-04-05', u'BUY', u'MSOFT', 1000, 72.0)
>>>
```

For more information about the SQL dialect supported by SQLite, see http://www.sqlite.org.

See Also:

http://www.pysqlite.org
: The pysqlite web page.

http://www.sqlite.org
: The SQLite web page; the documentation describes the syntax and the available data types for the supported SQL dialect.

../lib/module-sqlite3.html
: The documentation for the `sqlite3` module.

PEP 249, "*Database API Specification 2.0*"
: PEP written by Marc-André Lemburg.

13.5 The wsgiref package

The Web Server Gateway Interface (WSGI) v1.0 defines a standard interface between web servers and Python web applications and is described in PEP 333. The `wsgiref` package is a reference implementation of the WSGI specification.

The package includes a basic HTTP server that will run a WSGI application; this server is useful for debugging but isn't intended for production use. Setting up a server takes only a few lines of code:

```
from wsgiref import simple_server

wsgi_app = ...

host = ''
port = 8000
httpd = simple_server.make_server(host, port, wsgi_app)
httpd.serve_forever()
```

See Also:

http://www.wsgi.org
A central web site for WSGI-related resources.

PEP 333, "*Python Web Server Gateway Interface v1.0*"
PEP written by Phillip J. Eby.

14 Build and C API Changes

Changes to Python's build process and to the C API include:

- The Python source tree was converted from CVS to Subversion, in a complex migration procedure that was supervised and flawlessly carried out by Martin von Löwis. The procedure was developed as PEP 347.
- Coverity, a company that markets a source code analysis tool called Prevent, provided the results of their examination of the Python source code. The analysis found about 60 bugs that were quickly fixed. Many of the bugs were refcounting problems, often occurring in error-handling code. See http://scan.coverity.com for the statistics.
- The largest change to the C API came from PEP 353, which modifies the interpreter to use a `Py_ssize_t` type definition instead of `int`. See the earlier section 10 for a discussion of this change.
- The design of the bytecode compiler has changed a great deal, no longer generating bytecode by traversing the parse tree. Instead the parse tree is converted to an abstract syntax tree (or AST), and it is the abstract syntax tree that's traversed to produce the bytecode.

  It's possible for Python code to obtain AST objects by using the `compile()` built-in and specifying `_ast.PyCF_ONLY_AST` as the value of the *flags* parameter:

  ```
  from _ast import PyCF_ONLY_AST
  ast = compile("""a=0
  for i in range(10):
      a += i
  """, "<string>", 'exec', PyCF_ONLY_AST)

  assignment = ast.body[0]
  for_loop = ast.body[1]
  ```

  No official documentation has been written for the AST code yet, but PEP 339 discusses the design. To start learning about the code, read the definition of the various AST nodes in 'Parser/Python.asdl'. A Python script reads this file and generates a set of C structure definitions in 'Include/Python-ast.h'. The `PyParser_ASTFromString()` and `PyParser_ASTFromFile()`, defined in 'Include/pythonrun.h', take Python source as input and return the root of an AST representing the contents. This AST can then be turned into a code object by `PyAST_Compile()`. For more information, read the source code, and then ask questions on python-dev.

The AST code was developed under Jeremy Hylton's management, and implemented by (in alphabetical order) Brett Cannon, Nick Coghlan, Grant Edwards, John Ehresman, Kurt Kaiser, Neal Norwitz, Tim Peters, Armin Rigo, and Neil Schemenauer, plus the participants in a number of AST sprints at conferences such as PyCon.

- Evan Jones's patch to obmalloc, first described in a talk at PyCon DC 2005, was applied. Python 2.4 allocated small objects in 256K-sized arenas, but never freed arenas. With this patch, Python will free arenas when they're empty. The net effect is that on some platforms, when you allocate many objects, Python's memory usage may actually drop when you delete them and the memory may be returned to the operating system. (Implemented by Evan Jones, and reworked by Tim Peters.)

  Note that this change means extension modules must be more careful when allocating memory. Python's API has many different functions for allocating memory that are grouped into families. For example, `PyMem_Malloc()`, `PyMem_Realloc()`, and `PyMem_Free()` are one family that allocates raw memory, while `PyObject_Malloc()`, `PyObject_Realloc()`, and `PyObject_Free()` are another family that's supposed to be used for creating Python objects.

  Previously these different families all reduced to the platform's `malloc()` and `free()` functions. This meant it didn't matter if you got things wrong and allocated memory with the `PyMem` function but freed it with the `PyObject` function. With 2.5's changes to obmalloc, these families now do different things and mismatches will probably result in a segfault. You should carefully test your C extension modules with Python 2.5.

- The built-in set types now have an official C API. Call `PySet_New()` and `PyFrozenSet_New()` to create a new set, `PySet_Add()` and `PySet_Discard()` to add and remove elements, and `PySet_Contains` and `PySet_Size` to examine the set's state. (Contributed by Raymond Hettinger.)

- C code can now obtain information about the exact revision of the Python interpreter by calling the `Py_GetBuildInfo()` function that returns a string of build information like this: `"trunk:45355:45356M, Apr 13 2006, 07:42:19"`. (Contributed by Barry Warsaw.)

- Two new macros can be used to indicate C functions that are local to the current file so that a faster calling convention can be used. `Py_LOCAL(`*type*`)` declares the function as returning a value of the specified *type* and uses a fast-calling qualifier. `Py_LOCAL_INLINE(`*type*`)` does the same thing and also requests the function be inlined. If `PY_LOCAL_AGGRESSIVE` is defined before 'python.h' is included, a set of more aggressive optimizations are enabled for the module; you should benchmark the results to find out if these optimizations actually make the code faster. (Contributed by Fredrik Lundh at the NeedForSpeed sprint.)

- `PyErr_NewException(`*name*`, `*base*`, `*dict*`)` can now accept a tuple of base classes as its *base* argument. (Contributed by Georg Brandl.)

- The `PyErr_Warn()` function for issuing warnings is now deprecated in favour of `PyErr_WarnEx(category, message, stacklevel)` which lets you specify the number of stack frames separating this function and the caller. A *stacklevel* of 1 is the function calling `PyErr_WarnEx()`, 2 is the function above that, and so forth. (Added by Neal Norwitz.)

- The CPython interpreter is still written in C, but the code can now be compiled with a C++ compiler without errors. (Implemented by Anthony Baxter, Martin von Löwis, Skip Montanaro.)

- The `PyRange_New()` function was removed. It was never documented, never used in the core code, and had dangerously lax error checking. In the unlikely case that your extensions were using it, you can replace it by something like the following:

```
range = PyObject_CallFunction((PyObject*) &PyRange_Type, "lll",
                              start, stop, step);
```

14.1 Port-Specific Changes

- MacOS X (10.3 and higher): dynamic loading of modules now uses the `dlopen()` function instead of MacOS-specific functions.
- MacOS X: a **--enable-universalsdk** switch was added to the **configure** script that compiles the interpreter as a universal binary able to run on both PowerPC and Intel processors. (Contributed by Ronald Oussoren.)
- Windows: '.dll' is no longer supported as a filename extension for extension modules. '.pyd' is now the only filename extension that will be searched for.

15 Porting to Python 2.5

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

- ASCII is now the default encoding for modules. It's now a syntax error if a module contains string literals with 8-bit characters but doesn't have an encoding declaration. In Python 2.4 this triggered a warning, not a syntax error.
- Previously, the `gi_frame` attribute of a generator was always a frame object. Because of the PEP 342 changes described in section 7, it's now possible for `gi_frame` to be `None`.
- A new warning, `UnicodeWarning`, is triggered when you attempt to compare a Unicode string and an 8-bit string that can't be converted to Unicode using the default ASCII encoding. Previously such comparisons would raise a `UnicodeDecodeError` exception.
- Library: the `csv` module is now stricter about multi-line quoted fields. If your files contain newlines embedded within fields, the input should be split into lines in a manner which preserves the newline characters.
- Library: the `locale` module's `format()` function's would previously accept any string as long as no more than one %char specifier appeared. In Python 2.5, the argument must be exactly one %char specifier with no surrounding text.
- Library: The `pickle` and `cPickle` modules no longer accept a return value of `None` from the `__reduce__()` method; the method must return a tuple of arguments instead. The modules also no longer accept the deprecated *bin* keyword parameter.
- Library: The `SimpleXMLRPCServer` and `DocXMLRPCServer` classes now have a `rpc_paths` attribute that constrains XML-RPC operations to a limited set of URL paths; the default is to allow only `'/'` and `'/RPC2'`. Setting `rpc_paths` to `None` or an empty tuple disables this path checking.
- C API: Many functions now use `Py_ssize_t` instead of `int` to allow processing more data on 64-bit machines. Extension code may need to make the same change to avoid warnings and to support 64-bit machines. See the earlier section 10 for a discussion of this change.
- C API: The obmalloc changes mean that you must be careful to not mix usage of the `PyMem_*()` and `PyObject_*()` families of functions. Memory allocated with one family's `*_Malloc()` must be freed with the corresponding family's `*_Free()` function.

16 Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Nick Coghlan, Phillip J. Eby, Lars Gustäbel, Raymond Hettinger, Ralf W. Grosse-Kunstleve, Kent Johnson, Iain Lowe, Martin von Löwis, Fredrik Lundh, Andrew McNamara, Skip Montanaro, Gustavo Niemeyer, Paul Prescod, James Pryor, Mike Rovner, Scott Weikart, Barry Warsaw, Thomas Wouters.