What's New in Python

Release 2.6.5

A. M. Kuchling

March 19, 2010

Python Software Foundation
Email: docs@python.org

Contents

Author A.M. Kuchling (amk at amk.ca)

Release 2.6.5

Date March 19, 2010

This article explains the new features in Python 2.6, released on October 1 2008. The release schedule is described in **PEP 361**.

The major theme of Python 2.6 is preparing the migration path to Python 3.0, a major redesign of the language. Whenever possible, Python 2.6 incorporates new features and syntax from 3.0 while remaining compatible with existing code by not removing older features or syntax. When it's not possible to do that, Python 2.6 tries to do what it can, adding compatibility functions in a `future_builtins` module and a *-3* switch to warn about usages that will become unsupported in 3.0.

Some significant new packages have been added to the standard library, such as the `multiprocessing` and `json` modules, but there aren't many new features that aren't related to Python 3.0 in some way.

Python 2.6 also sees a number of improvements and bugfixes throughout the source. A search through the change logs finds there were 259 patches applied and 612 bugs fixed between Python 2.5 and 2.6. Both figures are likely to be underestimates.

This article doesn't attempt to provide a complete specification of the new features, but instead provides a convenient overview. For full details, you should refer to the documentation for Python 2.6. If you want to understand the rationale for the design and implementation, refer to the PEP for a particular new feature. Whenever possible, "What's New in Python" links to the bug/patch item for each change.

1 Python 3.0

The development cycle for Python versions 2.6 and 3.0 was synchronized, with the alpha and beta releases for both versions being made on the same days. The development of 3.0 has influenced many features in 2.6.

Python 3.0 is a far-ranging redesign of Python that breaks compatibility with the 2.x series. This means that existing Python code will need some conversion in order to run on Python 3.0. However, not all the changes in 3.0 necessarily break compatibility. In cases where new features won't cause existing code to break, they've been backported to 2.6 and are described in this document in the appropriate place. Some of the 3.0-derived features are:

- A `__complex__()` method for converting objects to a complex number.
- Alternate syntax for catching exceptions: `except TypeError as exc`.

- The addition of `functools.reduce()` as a synonym for the built-in `reduce()` function.

Python 3.0 adds several new built-in functions and changes the semantics of some existing built-ins. Functions that are new in 3.0 such as `bin()` have simply been added to Python 2.6, but existing built-ins haven't been changed; instead, the `future_builtins` module has versions with the new 3.0 semantics. Code written to be compatible with 3.0 can do `from future_builtins import hex, map` as necessary.

A new command-line switch, *-3*, enables warnings about features that will be removed in Python 3.0. You can run code with this switch to see how much work will be necessary to port code to 3.0. The value of this switch is available to Python code as the boolean variable `sys.py3kwarning`, and to C extension code as `Py_Py3kWarningFlag`.

See Also:

The 3xxx series of PEPs, which contains proposals for Python 3.0. **PEP 3000** describes the development process for Python 3.0. Start with **PEP 3100** that describes the general goals for Python 3.0, and then explore the higher-numbered PEPS that propose specific features.

2 Changes to the Development Process

While 2.6 was being developed, the Python development process underwent two significant changes: we switched from SourceForge's issue tracker to a customized Roundup installation, and the documentation was converted from LaTeX to reStructuredText.

2.1 New Issue Tracker: Roundup

For a long time, the Python developers had been growing increasingly annoyed by SourceForge's bug tracker. SourceForge's hosted solution doesn't permit much customization; for example, it wasn't possible to customize the life cycle of issues.

The infrastructure committee of the Python Software Foundation therefore posted a call for issue trackers, asking volunteers to set up different products and import some of the bugs and patches from SourceForge. Four different trackers were examined: Jira, Launchpad, Roundup, and Trac. The committee eventually settled on Jira and Roundup as the two candidates. Jira is a commercial product that offers no-cost hosted instances to free-software projects; Roundup is an open-source project that requires volunteers to administer it and a server to host it.

After posting a call for volunteers, a new Roundup installation was set up at http://bugs.python.org. One installation of Roundup can host multiple trackers, and this server now also hosts issue trackers for Jython and for the Python web site. It will surely find other uses in the future. Where possible, this edition of "What's New in Python" links to the bug/patch item for each change.

Hosting of the Python bug tracker is kindly provided by Upfront Systems of Stellenbosch, South Africa. Martin von Loewis put a lot of effort into importing existing bugs and patches from SourceForge; his scripts for this import operation are at http://svn.python.org/view/tracker/importer/ and may be useful to other projects wishing to move from SourceForge to Roundup.

See Also:

http://bugs.python.org The Python bug tracker.

http://bugs.jython.org: The Jython bug tracker.

http://roundup.sourceforge.net/ Roundup downloads and documentation.

http://svn.python.org/view/tracker/importer/ Martin von Loewis's conversion scripts.

2.2 New Documentation Format: reStructuredText Using Sphinx

The Python documentation was written using LaTeX since the project started around 1989. In the 1980s and early 1990s, most documentation was printed out for later study, not viewed online. LaTeX was widely used because

it provided attractive printed output while remaining straightforward to write once the basic rules of the markup were learned.

Today LaTeX is still used for writing publications destined for printing, but the landscape for programming tools has shifted. We no longer print out reams of documentation; instead, we browse through it online and HTML has become the most important format to support. Unfortunately, converting LaTeX to HTML is fairly complicated and Fred L. Drake Jr., the long-time Python documentation editor, spent a lot of time maintaining the conversion process. Occasionally people would suggest converting the documentation into SGML and later XML, but performing a good conversion is a major task and no one ever committed the time required to finish the job.

During the 2.6 development cycle, Georg Brandl put a lot of effort into building a new toolchain for processing the documentation. The resulting package is called Sphinx, and is available from http://sphinx.pocoo.org/.

Sphinx concentrates on HTML output, producing attractively styled and modern HTML; printed output is still supported through conversion to LaTeX. The input format is reStructuredText, a markup syntax supporting custom extensions and directives that is commonly used in the Python community.

Sphinx is a standalone package that can be used for writing, and almost two dozen other projects (listed on the Sphinx web site) have adopted Sphinx as their documentation tool.

See Also:

***Documenting Python* (in *Documenting Python*)** Describes how to write for Python's documentation.

Sphinx Documentation and code for the Sphinx toolchain.

Docutils The underlying reStructuredText parser and toolset.

3 PEP 343: The 'with' statement

The previous version, Python 2.5, added the '`with`' statement as an optional feature, to be enabled by a `from __future__ import with_statement` directive. In 2.6 the statement no longer needs to be specially enabled; this means that `with` is now always a keyword. The rest of this section is a copy of the corresponding section from the "What's New in Python 2.5" document; if you're familiar with the '`with`' statement from Python 2.5, you can skip this section.

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 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.

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 `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()
```

3.1 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 context manager's `__exit__()` method is called with three arguments, the exception details (`type, value, traceback`, the same values returned by `sys.exc_info()`, which can also be `None` if no exception occurred). The method's return value controls whether an 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
```

3.2 The contextlib module

The `contextlib` module provides some functions and a decorator that are useful when 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.

Using this decorator, our database example from the previous section could be written 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()` function returns its argument so that it can be bound to a variable, and calls the argument's `.close()` method 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.

The documentation for the `contextlib` module.

4 PEP 366: Explicit Relative Imports From a Main Module

Python's *-m* switch allows running a module as a script. When you ran a module that was located inside a package, relative imports didn't work correctly.

The fix for Python 2.6 adds a `__package__` attribute to modules. When this attribute is present, relative imports will be relative to the value of this attribute instead of the `__name__` attribute.

PEP 302-style importers can then set `__package__` as necessary. The `runpy` module that implements the *-m* switch now does this, so relative imports will now work correctly in scripts running from inside a package.

5 PEP 370: Per-user `site-packages` Directory

When you run Python, the module search path `sys.path` usually includes a directory whose path ends in `"site-packages"`. This directory is intended to hold locally-installed packages available to all users using a machine or a particular site installation.

Python 2.6 introduces a convention for user-specific site directories. The directory varies depending on the platform:

- Unix and Mac OS X: `~/.local/`

- Windows: `%APPDATA%/Python`

Within this directory, there will be version-specific subdirectories, such as `lib/python2.6/site-packages` on Unix/Mac OS and `Python26/site-packages` on Windows.

If you don't like the default directory, it can be overridden by an environment variable. **PYTHONUSERBASE** sets the root directory used for all Python versions supporting this feature. On Windows, the directory for application-specific data can be changed by setting the **APPDATA** environment variable. You can also modify the `site.py` file for your Python installation.

The feature can be disabled entirely by running Python with the *-s* option or setting the **PYTHONNOUSERSITE** environment variable.

See Also:

PEP 370 - Per-user `site-packages` Directory PEP written and implemented by Christian Heimes.

6 PEP 371: The `multiprocessing` Package

The new `multiprocessing` package lets Python programs create new processes that will perform a computation and return a result to the parent. The parent and child processes can communicate using queues and pipes, synchronize their operations using locks and semaphores, and can share simple arrays of data.

The `multiprocessing` module started out as an exact emulation of the `threading` module using processes instead of threads. That goal was discarded along the path to Python 2.6, but the general approach of the module is still similar. The fundamental class is the `Process`, which is passed a callable object and a collection of arguments. The `start()` method sets the callable running in a subprocess, after which you can call the `is_alive()` method to check whether the subprocess is still running and the `join()` method to wait for the process to exit.

Here's a simple example where the subprocess will calculate a factorial. The function doing the calculation is written strangely so that it takes significantly longer when the input argument is a multiple of 4.

```
import time
from multiprocessing import Process, Queue


def factorial(queue, N):
    "Compute a factorial."
    # If N is a multiple of 4, this function will take much longer.
    if (N % 4) == 0:
        time.sleep(.05 * N/4)

    # Calculate the result
    fact = 1L
    for i in range(1, N+1):
        fact = fact * i

    # Put the result on the queue
    queue.put(fact)

if __name__ == '__main__':
    queue = Queue()

    N = 5

    p = Process(target=factorial, args=(queue, N))
    p.start()
    p.join()
```

```
    result = queue.get()
    print 'Factorial', N, '=', result
```

A `Queue` is used to communicate the input parameter *N* and the result. The `Queue` object is stored in a global variable. The child process will use the value of the variable when the child was created; because it's a `Queue`, parent and child can use the object to communicate. (If the parent were to change the value of the global variable, the child's value would be unaffected, and vice versa.)

Two other classes, `Pool` and `Manager`, provide higher-level interfaces. `Pool` will create a fixed number of worker processes, and requests can then be distributed to the workers by calling `apply()` or `apply_async()` to add a single request, and `map()` or `map_async()` to add a number of requests. The following code uses a `Pool` to spread requests across 5 worker processes and retrieve a list of results:

```
from multiprocessing import Pool

def factorial(N, dictionary):
    "Compute a factorial."
    ...
p = Pool(5)
result = p.map(factorial, range(1, 1000, 10))
for v in result:
    print v
```

This produces the following output:

```
1
39916800
51090942171709440000
8222838654177922817725562880000000
334525266131638071081700620534407516651520000000000
...
```

The other high-level interface, the `Manager` class, creates a separate server process that can hold master copies of Python data structures. Other processes can then access and modify these data structures using proxy objects. The following example creates a shared dictionary by calling the `dict()` method; the worker processes then insert values into the dictionary. (Locking is not done for you automatically, which doesn't matter in this example. `Manager`'s methods also include `Lock()`, `RLock()`, and `Semaphore()` to create shared locks.)

```
import time
from multiprocessing import Pool, Manager

def factorial(N, dictionary):
    "Compute a factorial."
    # Calculate the result
    fact = 1L
    for i in range(1, N+1):
        fact = fact * i

    # Store result in dictionary
    dictionary[N] = fact

if __name__ == '__main__':
    p = Pool(5)
    mgr = Manager()
    d = mgr.dict()         # Create shared dictionary

    # Run tasks using the pool
    for N in range(1, 1000, 10):
        p.apply_async(factorial, (N, d))

    # Mark pool as closed -- no more tasks can be added.
    p.close()
```

```
    # Wait for tasks to exit
    p.join()

    # Output results
    for k, v in sorted(d.items()):
        print k, v
```

This will produce the output:

```
1 1
11 39916800
21 5109094217170944000
31 822283865417792281772556288000000
41 33452526613163807108170062053440751665152000000000
51 1551118753287382280224243016469303211063259720016986112000...
```

See Also:

The documentation for the `multiprocessing` module.

PEP 371 - Addition of the multiprocessing package PEP written by Jesse Noller and Richard Oudkerk; implemented by Richard Oudkerk and Jesse Noller.

7 PEP 3101: Advanced String Formatting

In Python 3.0, the % operator is supplemented by a more powerful string formatting method, `format()`. Support for the `str.format()` method has been backported to Python 2.6.

In 2.6, both 8-bit and Unicode strings have a *.format()* method that treats the string as a template and takes the arguments to be formatted. The formatting template uses curly brackets (*{*, *}*) as special characters:

```
>>> # Substitute positional argument 0 into the string.
>>> "User ID: {0}".format("root")
'User ID: root'
>>> # Use the named keyword arguments
>>> "User ID: {uid}   Last seen: {last_login}".format(
...     uid="root",
...     last_login = "5 Mar 2008 07:20")
'User ID: root   Last seen: 5 Mar 2008 07:20'
```

Curly brackets can be escaped by doubling them:

```
>>> "Empty dict: {{}}".format()
"Empty dict: {}"
```

Field names can be integers indicating positional arguments, such as `{0}`, `{1}`, etc. or names of keyword arguments. You can also supply compound field names that read attributes or access dictionary keys:

```
>>> import sys
>>> print 'Platform: {0.platform}\nPython version: {0.version}'.format(sys)
Platform: darwin
Python version: 2.6a1+ (trunk:61261M, Mar  5 2008, 20:29:41)
[GCC 4.0.1 (Apple Computer, Inc. build 5367)]'

>>> import mimetypes
>>> 'Content-type: {0[.mp4]}'.format(mimetypes.types_map)
'Content-type: video/mp4'
```

Note that when using dictionary-style notation such as `[.mp4]`, you don't need to put any quotation marks around the string; it will look up the value using `.mp4` as the key. Strings beginning with a number will be converted to an integer. You can't write more complicated expressions inside a format string.

So far we've shown how to specify which field to substitute into the resulting string. The precise formatting used is also controllable by adding a colon followed by a format specifier. For example:

```
>>> # Field 0: left justify, pad to 15 characters
>>> # Field 1: right justify, pad to 6 characters
>>> fmt = '{0:15} ${1:>6}'
>>> fmt.format('Registration', 35)
'Registration    $    35'
>>> fmt.format('Tutorial', 50)
'Tutorial        $    50'
>>> fmt.format('Banquet', 125)
'Banquet         $   125'
```

Format specifiers can reference other fields through nesting:

```
>>> fmt = '{0:{1}}'
>>> width = 15
>>> fmt.format('Invoice #1234', width)
'Invoice #1234  '
>>> width = 35
>>> fmt.format('Invoice #1234', width)
'Invoice #1234                      '
```

The alignment of a field within the desired width can be specified:

Character	Effect
< (default)	Left-align
>	Right-align
^	Center
=	(For numeric types only) Pad after the sign.

Format specifiers can also include a presentation type, which controls how the value is formatted. For example, floating-point numbers can be formatted as a general number or in exponential notation:

```
>>> '{0:g}'.format(3.75)
'3.75'
>>> '{0:e}'.format(3.75)
'3.750000e+00'
```

A variety of presentation types are available. Consult the 2.6 documentation for a *complete list* (in *The Python Library Reference*); here's a sample:

b	Binary. Outputs the number in base 2.
c	Character. Converts the integer to the corresponding Unicode character before printing.
d	Decimal Integer. Outputs the number in base 10.
o	Octal format. Outputs the number in base 8.
x	Hex format. Outputs the number in base 16, using lower-case letters for the digits above 9.
e	Exponent notation. Prints the number in scientific notation using the letter 'e' to indicate the exponent.
g	General format. This prints the number as a fixed-point number, unless the number is too large, in which case it switches to 'e' exponent notation.
n	Number. This is the same as 'g' (for floats) or 'd' (for integers), except that it uses the current locale setting to insert the appropriate number separator characters.
%	Percentage. Multiplies the number by 100 and displays in fixed ('f') format, followed by a percent sign.

Classes and types can define a `__format__()` method to control how they're formatted. It receives a single argument, the format specifier:

```
def __format__(self, format_spec):
    if isinstance(format_spec, unicode):
        return unicode(str(self))
    else:
        return str(self)
```

There's also a `format()` built-in that will format a single value. It calls the type's `__format__()` method with the provided specifier:

```
>>> format(75.6564, '.2f')
'75.66'
```

See Also:

***Format String Syntax* (in *The Python Library Reference*)** The reference documentation for format fields.

PEP 3101 - Advanced String Formatting PEP written by Talin. Implemented by Eric Smith.

8 PEP 3105: `print` As a Function

The `print` statement becomes the `print()` function in Python 3.0. Making `print()` a function makes it possible to replace the function by doing `def print(...)` or importing a new function from somewhere else.

Python 2.6 has a `__future__` import that removes `print` as language syntax, letting you use the functional form instead. For example:

```
>>> from __future__ import print_function
>>> print('# of entries', len(dictionary), file=sys.stderr)
```

The signature of the new function is:

```
def print(*args, sep=' ', end='\n', file=None)
```

The parameters are:

- *args*: positional arguments whose values will be printed out.
- *sep*: the separator, which will be printed between arguments.
- *end*: the ending text, which will be printed after all of the arguments have been output.
- *file*: the file object to which the output will be sent.

See Also:

PEP 3105 - Make print a function PEP written by Georg Brandl.

9 PEP 3110: Exception-Handling Changes

One error that Python programmers occasionally make is writing the following code:

```
try:
    ...
except TypeError, ValueError:  # Wrong!
    ...
```

The author is probably trying to catch both `TypeError` and `ValueError` exceptions, but this code actually does something different: it will catch `TypeError` and bind the resulting exception object to the local name `"ValueError"`. The `ValueError` exception will not be caught at all. The correct code specifies a tuple of exceptions:

```
try:
    ...
except (TypeError, ValueError):
    ...
```

This error happens because the use of the comma here is ambiguous: does it indicate two different nodes in the parse tree, or a single node that's a tuple?

Python 3.0 makes this unambiguous by replacing the comma with the word "as". To catch an exception and store the exception object in the variable `exc`, you must write:

```
try:
    ...
except TypeError as exc:
    ...
```

Python 3.0 will only support the use of "as", and therefore interprets the first example as catching two different exceptions. Python 2.6 supports both the comma and "as", so existing code will continue to work. We therefore suggest using "as" when writing new Python code that will only be executed with 2.6.

See Also:

PEP 3110 - Catching Exceptions in Python 3000 PEP written and implemented by Collin Winter.

10 PEP 3112: Byte Literals

Python 3.0 adopts Unicode as the language's fundamental string type and denotes 8-bit literals differently, either as `b'string'` or using a `bytes` constructor. For future compatibility, Python 2.6 adds `bytes` as a synonym for the `str` type, and it also supports the `b""` notation.

The 2.6 `str` differs from 3.0's `bytes` type in various ways; most notably, the constructor is completely different. In 3.0, `bytes([65, 66, 67])` is 3 elements long, containing the bytes representing `ABC`; in 2.6, `bytes([65, 66, 67])` returns the 12-byte string representing the `str()` of the list.

The primary use of `bytes` in 2.6 will be to write tests of object type such as `isinstance(x, bytes)`. This will help the 2to3 converter, which can't tell whether 2.x code intends strings to contain either characters or 8-bit bytes; you can now use either `bytes` or `str` to represent your intention exactly, and the resulting code will also be correct in Python 3.0.

There's also a `__future__` import that causes all string literals to become Unicode strings. This means that `\u` escape sequences can be used to include Unicode characters:

```
from __future__ import unicode_literals

s = ('\u751f\u3080\u304e\u3000\u751f\u3054'
     '\u3081\u3000\u751f\u305f\u307e\u3054')

print len(s)                # 12 Unicode characters
```

At the C level, Python 3.0 will rename the existing 8-bit string type, called `PyStringObject` in Python 2.x, to `PyBytesObject`. Python 2.6 uses `#define` to support using the names `PyBytesObject()`, `PyBytes_Check()`, `PyBytes_FromStringAndSize()`, and all the other functions and macros used with strings.

Instances of the `bytes` type are immutable just as strings are. A new `bytearray` type stores a mutable sequence of bytes:

```
>>> bytearray([65, 66, 67])
bytearray(b'ABC')
>>> b = bytearray(u'\u21ef\u3244', 'utf-8')
>>> b
bytearray(b'\xe2\x87\xaf\xe3\x89\x84')
>>> b[0] = '\xe3'
>>> b
bytearray(b'\xe3\x87\xaf\xe3\x89\x84')
>>> unicode(str(b), 'utf-8')
u'\u31ef \u3244'
```

Byte arrays support most of the methods of string types, such as `startswith()`/`endswith()`, `find()`/`rfind()`, and some of the methods of lists, such as `append()`, `pop()`, and `reverse()`.

```
>>> b = bytearray('ABC')
>>> b.append('d')
```

```
>>> b.append(ord('e'))
>>> b
bytearray(b'ABCde')
```

There's also a corresponding C API, with `PyByteArray_FromObject()`, `PyByteArray_FromStringAndSize()`, and various other functions.

See Also:

PEP 3112 - Bytes literals in Python 3000 PEP written by Jason Orendorff; backported to 2.6 by Christian Heimes.

11 PEP 3116: New I/O Library

Python's built-in file objects support a number of methods, but file-like objects don't necessarily support all of them. Objects that imitate files usually support `read()` and `write()`, but they may not support `readline()`, for example. Python 3.0 introduces a layered I/O library in the `io` module that separates buffering and text-handling features from the fundamental read and write operations.

There are three levels of abstract base classes provided by the `io` module:

- `RawIOBase` defines raw I/O operations: `read()`, `readinto()`, `write()`, `seek()`, `tell()`, `truncate()`, and `close()`. Most of the methods of this class will often map to a single system call. There are also `readable()`, `writable()`, and `seekable()` methods for determining what operations a given object will allow.

  Python 3.0 has concrete implementations of this class for files and sockets, but Python 2.6 hasn't restructured its file and socket objects in this way.

- `BufferedIOBase` is an abstract base class that buffers data in memory to reduce the number of system calls used, making I/O processing more efficient. It supports all of the methods of `RawIOBase`, and adds a `raw` attribute holding the underlying raw object.

  There are five concrete classes implementing this ABC. `BufferedWriter` and `BufferedReader` are for objects that support write-only or read-only usage that have a `seek()` method for random access. `BufferedRandom` objects support read and write access upon the same underlying stream, and `BufferedRWPair` is for objects such as TTYs that have both read and write operations acting upon unconnected streams of data. The `BytesIO` class supports reading, writing, and seeking over an in-memory buffer.

- `TextIOBase`: Provides functions for reading and writing strings (remember, strings will be Unicode in Python 3.0), and supporting universal newlines. `TextIOBase` defines the `readline()` method and supports iteration upon objects.

  There are two concrete implementations. `TextIOWrapper` wraps a buffered I/O object, supporting all of the methods for text I/O and adding a `buffer` attribute for access to the underlying object. `StringIO` simply buffers everything in memory without ever writing anything to disk.

  (In Python 2.6, `io.StringIO` is implemented in pure Python, so it's pretty slow. You should therefore stick with the existing `StringIO` module or `cStringIO` for now. At some point Python 3.0's `io` module will be rewritten into C for speed, and perhaps the C implementation will be backported to the 2.x releases.)

In Python 2.6, the underlying implementations haven't been restructured to build on top of the `io` module's classes. The module is being provided to make it easier to write code that's forward-compatible with 3.0, and to save developers the effort of writing their own implementations of buffering and text I/O.

See Also:

PEP 3116 - New I/O PEP written by Daniel Stutzbach, Mike Verdone, and Guido van Rossum. Code by Guido van Rossum, Georg Brandl, Walter Doerwald, Jeremy Hylton, Martin von Loewis, Tony Lownds, and others.

12 PEP 3118: Revised Buffer Protocol

The buffer protocol is a C-level API that lets Python types exchange pointers into their internal representations. A memory-mapped file can be viewed as a buffer of characters, for example, and this lets another module such as `re` treat memory-mapped files as a string of characters to be searched.

The primary users of the buffer protocol are numeric-processing packages such as NumPy, which expose the internal representation of arrays so that callers can write data directly into an array instead of going through a slower API. This PEP updates the buffer protocol in light of experience from NumPy development, adding a number of new features such as indicating the shape of an array or locking a memory region.

The most important new C API function is `PyObject_GetBuffer(PyObject *obj, Py_buffer *view, int flags)`, which takes an object and a set of flags, and fills in the `Py_buffer` structure with information about the object's memory representation. Objects can use this operation to lock memory in place while an external caller could be modifying the contents, so there's a corresponding `PyBuffer_Release(Py_buffer *view)` to indicate that the external caller is done.

The *flags* argument to `PyObject_GetBuffer()` specifies constraints upon the memory returned. Some examples are:

- `PyBUF_WRITABLE` indicates that the memory must be writable.
- `PyBUF_LOCK` requests a read-only or exclusive lock on the memory.
- `PyBUF_C_CONTIGUOUS` and `PyBUF_F_CONTIGUOUS` requests a C-contiguous (last dimension varies the fastest) or Fortran-contiguous (first dimension varies the fastest) array layout.

Two new argument codes for `PyArg_ParseTuple()`, `s*` and `z*`, return locked buffer objects for a parameter.

See Also:

PEP 3118 - Revising the buffer protocol PEP written by Travis Oliphant and Carl Banks; implemented by Travis Oliphant.

13 PEP 3119: Abstract Base Classes

Some object-oriented languages such as Java support interfaces, declaring that a class has a given set of methods or supports a given access protocol. Abstract Base Classes (or ABCs) are an equivalent feature for Python. The ABC support consists of an `abc` module containing a metaclass called `ABCMeta`, special handling of this metaclass by the `isinstance()` and `issubclass()` built-ins, and a collection of basic ABCs that the Python developers think will be widely useful. Future versions of Python will probably add more ABCs.

Let's say you have a particular class and wish to know whether it supports dictionary-style access. The phrase "dictionary-style" is vague, however. It probably means that accessing items with `obj[1]` works. Does it imply that setting items with `obj[2] = value` works? Or that the object will have `keys()`, `values()`, and `items()` methods? What about the iterative variants such as `iterkeys()`? `copy()` and `update()`? Iterating over the object with `iter()`?

The Python 2.6 `collections` module includes a number of different ABCs that represent these distinctions. `Iterable` indicates that a class defines `__iter__()`, and `Container` means the class defines a `__contains__()` method and therefore supports `x in y` expressions. The basic dictionary interface of getting items, setting items, and `keys()`, `values()`, and `items()`, is defined by the `MutableMapping` ABC.

You can derive your own classes from a particular ABC to indicate they support that ABC's interface:

```
import collections

class Storage(collections.MutableMapping):
    ...
```

Alternatively, you could write the class without deriving from the desired ABC and instead register the class by calling the ABC's `register()` method:

```
import collections

class Storage:
    ...

collections.MutableMapping.register(Storage)
```

For classes that you write, deriving from the ABC is probably clearer. The `register()` method is useful when you've written a new ABC that can describe an existing type or class, or if you want to declare that some third-party class implements an ABC. For example, if you defined a `PrintableType` ABC, it's legal to do:

```
# Register Python's types
PrintableType.register(int)
PrintableType.register(float)
PrintableType.register(str)
```

Classes should obey the semantics specified by an ABC, but Python can't check this; it's up to the class author to understand the ABC's requirements and to implement the code accordingly.

To check whether an object supports a particular interface, you can now write:

```
def func(d):
    if not isinstance(d, collections.MutableMapping):
        raise ValueError("Mapping object expected, not %r" % d)
```

Don't feel that you must now begin writing lots of checks as in the above example. Python has a strong tradition of duck-typing, where explicit type-checking is never done and code simply calls methods on an object, trusting that those methods will be there and raising an exception if they aren't. Be judicious in checking for ABCs and only do it where it's absolutely necessary.

You can write your own ABCs by using `abc.ABCMeta` as the metaclass in a class definition:

```
from abc import ABCMeta, abstractmethod

class Drawable():
    __metaclass__ = ABCMeta

    @abstractmethod
    def draw(self, x, y, scale=1.0):
        pass

    def draw_doubled(self, x, y):
        self.draw(x, y, scale=2.0)


class Square(Drawable):
    def draw(self, x, y, scale):
        ...
```

In the `Drawable` ABC above, the `draw_doubled()` method renders the object at twice its size and can be implemented in terms of other methods described in `Drawable`. Classes implementing this ABC therefore don't need to provide their own implementation of `draw_doubled()`, though they can do so. An implementation of `draw()` is necessary, though; the ABC can't provide a useful generic implementation.

You can apply the `@abstractmethod` decorator to methods such as `draw()` that must be implemented; Python will then raise an exception for classes that don't define the method. Note that the exception is only raised when you actually try to create an instance of a subclass lacking the method:

```
>>> class Circle(Drawable):
...     pass
...
>>> c = Circle()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
TypeError: Can't instantiate abstract class Circle with abstract methods draw
>>>
```

Abstract data attributes can be declared using the `@abstractproperty` decorator:

```
from abc import abstractproperty
...

@abstractproperty
def readonly(self):
   return self._x
```

Subclasses must then define a `readonly()` property.

See Also:

PEP 3119 - Introducing Abstract Base Classes PEP written by Guido van Rossum and Talin. Implemented by Guido van Rossum. Backported to 2.6 by Benjamin Aranguren, with Alex Martelli.

14 PEP 3127: Integer Literal Support and Syntax

Python 3.0 changes the syntax for octal (base-8) integer literals, prefixing them with "0o" or "0O" instead of a leading zero, and adds support for binary (base-2) integer literals, signalled by a "0b" or "0B" prefix.

Python 2.6 doesn't drop support for a leading 0 signalling an octal number, but it does add support for "0o" and "0b":

```
>>> 0o21, 2*8 + 1
(17, 17)
>>> 0b101111
47
```

The `oct()` built-in still returns numbers prefixed with a leading zero, and a new `bin()` built-in returns the binary representation for a number:

```
>>> oct(42)
'052'
>>> future_builtins.oct(42)
'0o52'
>>> bin(173)
'0b10101101'
```

The `int()` and `long()` built-ins will now accept the "0o" and "0b" prefixes when base-8 or base-2 are requested, or when the *base* argument is zero (signalling that the base used should be determined from the string):

```
>>> int ('0o52', 0)
42
>>> int('1101', 2)
13
>>> int('0b1101', 2)
13
>>> int('0b1101', 0)
13
```

See Also:

PEP 3127 - Integer Literal Support and Syntax PEP written by Patrick Maupin; backported to 2.6 by Eric Smith.

15 PEP 3129: Class Decorators

Decorators have been extended from functions to classes. It's now legal to write:

```
@foo
@bar
class A:
  pass
```

This is equivalent to:

```
class A:
  pass

A = foo(bar(A))
```

See Also:

PEP 3129 - Class Decorators PEP written by Collin Winter.

16 PEP 3141: A Type Hierarchy for Numbers

Python 3.0 adds several abstract base classes for numeric types inspired by Scheme's numeric tower. These classes were backported to 2.6 as the `numbers` module.

The most general ABC is `Number`. It defines no operations at all, and only exists to allow checking if an object is a number by doing `isinstance(obj, Number)`.

`Complex` is a subclass of `Number`. Complex numbers can undergo the basic operations of addition, subtraction, multiplication, division, and exponentiation, and you can retrieve the real and imaginary parts and obtain a number's conjugate. Python's built-in complex type is an implementation of `Complex`.

`Real` further derives from `Complex`, and adds operations that only work on real numbers: `floor()`, `trunc()`, rounding, taking the remainder mod N, floor division, and comparisons.

`Rational` numbers derive from `Real`, have `numerator` and `denominator` properties, and can be converted to floats. Python 2.6 adds a simple rational-number class, `Fraction`, in the `fractions` module. (It's called `Fraction` instead of `Rational` to avoid a name clash with `numbers.Rational`.)

`Integral` numbers derive from `Rational`, and can be shifted left and right with `<<` and `>>`, combined using bitwise operations such as `&` and `|`, and can be used as array indexes and slice boundaries.

In Python 3.0, the PEP slightly redefines the existing built-ins `round()`, `math.floor()`, `math.ceil()`, and adds a new one, `math.trunc()`, that's been backported to Python 2.6. `math.trunc()` rounds toward zero, returning the closest `Integral` that's between the function's argument and zero.

See Also:

PEP 3141 - A Type Hierarchy for Numbers PEP written by Jeffrey Yasskin.

Scheme's numerical tower, from the Guile manual.

Scheme's number datatypes from the R5RS Scheme specification.

16.1 The `fractions` Module

To fill out the hierarchy of numeric types, the `fractions` module provides a rational-number class. Rational numbers store their values as a numerator and denominator forming a fraction, and can exactly represent numbers such as `2/3` that floating-point numbers can only approximate.

The `Fraction` constructor takes two `Integral` values that will be the numerator and denominator of the resulting fraction.

```
>>> from fractions import Fraction
>>> a = Fraction(2, 3)
>>> b = Fraction(2, 5)
>>> float(a), float(b)
(0.6666666666666663, 0.40000000000000002)
>>> a+b
Fraction(16, 15)
>>> a/b
Fraction(5, 3)
```

For converting floating-point numbers to rationals, the float type now has an `as_integer_ratio()` method that returns the numerator and denominator for a fraction that evaluates to the same floating-point value:

```
>>> (2.5) .as_integer_ratio()
(5, 2)
>>> (3.1415) .as_integer_ratio()
(7074029114692207L, 2251799813685248L)
>>> (1./3) .as_integer_ratio()
(6004799503160661L, 18014398509481984L)
```

Note that values that can only be approximated by floating-point numbers, such as 1./3, are not simplified to the number being approximated; the fraction attempts to match the floating-point value **exactly**.

The `fractions` module is based upon an implementation by Sjoerd Mullender that was in Python's `Demo/classes/` directory for a long time. This implementation was significantly updated by Jeffrey Yasskin.

17 Other Language Changes

Some smaller changes made to the core Python language are:

- Directories and zip archives containing a `__main__.py` file can now be executed directly by passing their name to the interpreter. The directory or zip archive is automatically inserted as the first entry in sys.path. (Suggestion and initial patch by Andy Chu, subsequently revised by Phillip J. Eby and Nick Coghlan; issue 1739468.)
- The `hasattr()` function was catching and ignoring all errors, under the assumption that they meant a `__getattr__()` method was failing somehow and the return value of `hasattr()` would therefore be `False`. This logic shouldn't be applied to `KeyboardInterrupt` and `SystemExit`, however; Python 2.6 will no longer discard such exceptions when `hasattr()` encounters them. (Fixed by Benjamin Peterson; issue 2196.)
- When calling a function using the `**` syntax to provide keyword arguments, you are no longer required to use a Python dictionary; any mapping will now work:

  ```
  >>> def f(**kw):
  ...    print sorted(kw)
  ...
  >>> ud=UserDict.UserDict()
  >>> ud['a'] = 1
  >>> ud['b'] = 'string'
  >>> f(**ud)
  ['a', 'b']
  ```

  (Contributed by Alexander Belopolsky; issue 1686487.)

  It's also become legal to provide keyword arguments after a `*args` argument to a function call.

  ```
  >>> def f(*args, **kw):
  ...     print args, kw
  ...
  >>> f(1,2,3, *(4,5,6), keyword=13)
  (1, 2, 3, 4, 5, 6) {'keyword': 13}
  ```

Previously this would have been a syntax error. (Contributed by Amaury Forgeot d'Arc; issue 3473.)

- A new built-in, `next(iterator, [default])` returns the next item from the specified iterator. If the *default* argument is supplied, it will be returned if *iterator* has been exhausted; otherwise, the `StopIteration` exception will be raised. (Backported in issue 2719.)

- Tuples now have `index()` and `count()` methods matching the list type's `index()` and `count()` methods:

```
>>> t = (0,1,2,3,4,0,1,2)
>>> t.index(3)
3
>>> t.count(0)
2
```

(Contributed by Raymond Hettinger)

- The built-in types now have improved support for extended slicing syntax, accepting various combinations of `(start, stop, step)`. Previously, the support was partial and certain corner cases wouldn't work. (Implemented by Thomas Wouters.)

- Properties now have three attributes, `getter`, `setter` and `deleter`, that are decorators providing useful shortcuts for adding a getter, setter or deleter function to an existing property. You would use them like this:

```
class C(object):
    @property
    def x(self):
        return self._x

    @x.setter
    def x(self, value):
        self._x = value

    @x.deleter
    def x(self):
        del self._x

class D(C):
    @C.x.getter
    def x(self):
        return self._x * 2

    @x.setter
    def x(self, value):
        self._x = value / 2
```

- Several methods of the built-in set types now accept multiple iterables: `intersection()`, `intersection_update()`, `union()`, `update()`, `difference()` and `difference_update()`.

```
>>> s=set('1234567890')
>>> s.intersection('abc123', 'cdf246')  # Intersection between all inputs
set(['2'])
>>> s.difference('246', '789')
set(['1', '0', '3', '5'])
```

(Contributed by Raymond Hettinger.)

- Many floating-point features were added. The `float()` function will now turn the string `nan` into an IEEE 754 Not A Number value, and `+inf` and `-inf` into positive or negative infinity. This works on any platform with IEEE 754 semantics. (Contributed by Christian Heimes; issue 1635.)

  Other functions in the `math` module, `isinf()` and `isnan()`, return true if their floating-point argument is infinite or Not A Number. (issue 1640)

Conversion functions were added to convert floating-point numbers into hexadecimal strings (issue 3008). These functions convert floats to and from a string representation without introducing rounding errors from the conversion between decimal and binary. Floats have a `hex()` method that returns a string representation, and the `float.fromhex()` method converts a string back into a number:

```
>>> a = 3.75
>>> a.hex()
'0x1.e000000000000p+1'
>>> float.fromhex('0x1.e000000000000p+1')
3.75
>>> b=1./3
>>> b.hex()
'0x1.5555555555555p-2'
```

- A numerical nicety: when creating a complex number from two floats on systems that support signed zeros (-0 and +0), the `complex()` constructor will now preserve the sign of the zero. (Fixed by Mark T. Dickinson; issue 1507.)

- Classes that inherit a `__hash__()` method from a parent class can set `__hash__ = None` to indicate that the class isn't hashable. This will make `hash(obj)` raise a `TypeError` and the class will not be indicated as implementing the `Hashable` ABC.

  You should do this when you've defined a `__cmp__()` or `__eq__()` method that compares objects by their value rather than by identity. All objects have a default hash method that uses `id(obj)` as the hash value. There's no tidy way to remove the `__hash__()` method inherited from a parent class, so assigning `None` was implemented as an override. At the C level, extensions can set `tp_hash` to `PyObject_HashNotImplemented()`. (Fixed by Nick Coghlan and Amaury Forgeot d'Arc; issue 2235.)

- The `GeneratorExit` exception now subclasses `BaseException` instead of `Exception`. This means that an exception handler that does `except Exception:` will not inadvertently catch `GeneratorExit`. (Contributed by Chad Austin; issue 1537.)

- Generator objects now have a `gi_code` attribute that refers to the original code object backing the generator. (Contributed by Collin Winter; issue 1473257.)

- The `compile()` built-in function now accepts keyword arguments as well as positional parameters. (Contributed by Thomas Wouters; issue 1444529.)

- The `complex()` constructor now accepts strings containing parenthesized complex numbers, meaning that `complex(repr(cplx))` will now round-trip values. For example, `complex('(3+4j)')` now returns the value (3+4j). (issue 1491866)

- The string `translate()` method now accepts `None` as the translation table parameter, which is treated as the identity transformation. This makes it easier to carry out operations that only delete characters. (Contributed by Bengt Richter and implemented by Raymond Hettinger; issue 1193128.)

- The built-in `dir()` function now checks for a `__dir__()` method on the objects it receives. This method must return a list of strings containing the names of valid attributes for the object, and lets the object control the value that `dir()` produces. Objects that have `__getattr__()` or `__getattribute__()` methods can use this to advertise pseudo-attributes they will honor. (issue 1591665)

- Instance method objects have new attributes for the object and function comprising the method; the new synonym for `im_self` is `__self__`, and `im_func` is also available as `__func__`. The old names are still supported in Python 2.6, but are gone in 3.0.

- An obscure change: when you use the `locals()` function inside a `class` statement, the resulting dictionary no longer returns free variables. (Free variables, in this case, are variables referenced in the `class` statement that aren't attributes of the class.)

17.1 Optimizations

- The `warnings` module has been rewritten in C. This makes it possible to invoke warnings from the parser, and may also make the interpreter's startup faster. (Contributed by Neal Norwitz and Brett Cannon; issue 1631171.)
- Type objects now have a cache of methods that can reduce the work required to find the correct method implementation for a particular class; once cached, the interpreter doesn't need to traverse base classes to figure out the right method to call. The cache is cleared if a base class or the class itself is modified, so the cache should remain correct even in the face of Python's dynamic nature. (Original optimization implemented by Armin Rigo, updated for Python 2.6 by Kevin Jacobs; issue 1700288.)

  By default, this change is only applied to types that are included with the Python core. Extension modules may not necessarily be compatible with this cache, so they must explicitly add `Py_TPFLAGS_HAVE_VERSION_TAG` to the module's `tp_flags` field to enable the method cache. (To be compatible with the method cache, the extension module's code must not directly access and modify the `tp_dict` member of any of the types it implements. Most modules don't do this, but it's impossible for the Python interpreter to determine that. See issue 1878 for some discussion.)
- Function calls that use keyword arguments are significantly faster by doing a quick pointer comparison, usually saving the time of a full string comparison. (Contributed by Raymond Hettinger, after an initial implementation by Antoine Pitrou; issue 1819.)
- All of the functions in the `struct` module have been rewritten in C, thanks to work at the Need For Speed sprint. (Contributed by Raymond Hettinger.)
- Some of the standard built-in types now set a bit in their type objects. This speeds up checking whether an object is a subclass of one of these types. (Contributed by Neal Norwitz.)
- Unicode strings now use faster code for detecting whitespace and line breaks; this speeds up the `split()` method by about 25% and `splitlines()` by 35%. (Contributed by Antoine Pitrou.) Memory usage is reduced by using pymalloc for the Unicode string's data.
- The `with` statement now stores the `__exit__()` method on the stack, producing a small speedup. (Implemented by Jeffrey Yasskin.)
- To reduce memory usage, the garbage collector will now clear internal free lists when garbage-collecting the highest generation of objects. This may return memory to the operating system sooner.

17.2 Interpreter Changes

Two command-line options have been reserved for use by other Python implementations. The *-J* switch has been reserved for use by Jython for Jython-specific options, such as switches that are passed to the underlying JVM. *-X* has been reserved for options specific to a particular implementation of Python such as CPython, Jython, or IronPython. If either option is used with Python 2.6, the interpreter will report that the option isn't currently used.

Python can now be prevented from writing `.pyc` or `.pyo` files by supplying the *-B* switch to the Python interpreter, or by setting the **PYTHONDONTWRITEBYTECODE** environment variable before running the interpreter. This setting is available to Python programs as the `sys.dont_write_bytecode` variable, and Python code can change the value to modify the interpreter's behaviour. (Contributed by Neal Norwitz and Georg Brandl.)

The encoding used for standard input, output, and standard error can be specified by setting the **PYTHONIOENCODING** environment variable before running the interpreter. The value should be a string in the form `<encoding>` or `<encoding>:<errorhandler>`. The *encoding* part specifies the encoding's name, e.g. `utf-8` or `latin-1`; the optional *errorhandler* part specifies what to do with characters that can't be handled by the encoding, and should be one of "error", "ignore", or "replace". (Contributed by Martin von Loewis.)

18 New and Improved Modules

As in every release, 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 Subversion logs for all the details.

- The `asyncore` and `asynchat` modules are being actively maintained again, and a number of patches and bugfixes were applied. (Maintained by Josiah Carlson; see issue 1736190 for one patch.)

- The `bsddb` module also has a new maintainer, Jesús Cea, and the package is now available as a standalone package. The web page for the package is www.jcea.es/programacion/pybsddb.htm. The plan is to remove the package from the standard library in Python 3.0, because its pace of releases is much more frequent than Python's.

  The `bsddb.dbshelve` module now uses the highest pickling protocol available, instead of restricting itself to protocol 1. (Contributed by W. Barnes.)

- The `cgi` module will now read variables from the query string of an HTTP POST request. This makes it possible to use form actions with URLs that include query strings such as "/cgi-bin/add.py?category=1". (Contributed by Alexandre Fiori and Nubis; issue 1817.)

  The `parse_qs()` and `parse_qsl()` functions have been relocated from the `cgi` module to the `urlparse` module. The versions still available in the `cgi` module will trigger `PendingDeprecationWarning` messages in 2.6 (issue 600362).

- The `cmath` module underwent extensive revision, contributed by Mark Dickinson and Christian Heimes. Five new functions were added:

  - `polar()` converts a complex number to polar form, returning the modulus and argument of the complex number.
  - `rect()` does the opposite, turning a modulus, argument pair back into the corresponding complex number.
  - `phase()` returns the argument (also called the angle) of a complex number.
  - `isnan()` returns True if either the real or imaginary part of its argument is a NaN.
  - `isinf()` returns True if either the real or imaginary part of its argument is infinite.

  The revisions also improved the numerical soundness of the `cmath` module. For all functions, the real and imaginary parts of the results are accurate to within a few units of least precision (ulps) whenever possible. See issue 1381 for the details. The branch cuts for `asinh()`, `atanh()`: and `atan()` have also been corrected.

  The tests for the module have been greatly expanded; nearly 2000 new test cases exercise the algebraic functions.

  On IEEE 754 platforms, the `cmath` module now handles IEEE 754 special values and floating-point exceptions in a manner consistent with Annex 'G' of the C99 standard.

- A new data type in the `collections` module: `namedtuple(typename, fieldnames)` is a factory function that creates subclasses of the standard tuple whose fields are accessible by name as well as index. For example:

```
>>> var_type = collections.namedtuple('variable',
...                    'id name type size')
>>> # Names are separated by spaces or commas.
>>> # 'id, name, type, size' would also work.
>>> var_type._fields
('id', 'name', 'type', 'size')

>>> var = var_type(1, 'frequency', 'int', 4)
>>> print var[0], var.id     # Equivalent
1 1
>>> print var[2], var.type   # Equivalent
int int
>>> var._asdict()
{'size': 4, 'type': 'int', 'id': 1, 'name': 'frequency'}
>>> v2 = var._replace(name='amplitude')
```

```
>>> v2
variable(id=1, name='amplitude', type='int', size=4)
```

Several places in the standard library that returned tuples have been modified to return `namedtuple` instances. For example, the `Decimal.as_tuple()` method now returns a named tuple with `sign`, `digits`, and `exponent` fields.

(Contributed by Raymond Hettinger.)

- Another change to the `collections` module is that the `deque` type now supports an optional *maxlen* parameter; if supplied, the deque's size will be restricted to no more than *maxlen* items. Adding more items to a full deque causes old items to be discarded.

```
>>> from collections import deque
>>> dq=deque(maxlen=3)
>>> dq
deque([], maxlen=3)
>>> dq.append(1) ; dq.append(2) ; dq.append(3)
>>> dq
deque([1, 2, 3], maxlen=3)
>>> dq.append(4)
>>> dq
deque([2, 3, 4], maxlen=3)
```

(Contributed by Raymond Hettinger.)

- The `Cookie` module's `Morsel` objects now support an `httponly` attribute. In some browsers. cookies with this attribute set cannot be accessed or manipulated by JavaScript code. (Contributed by Arvin Schnell; issue 1638033.)

- A new window method in the `curses` module, `chgat()`, changes the display attributes for a certain number of characters on a single line. (Contributed by Fabian Kreutz.)

```
# Boldface text starting at y=0,x=21
# and affecting the rest of the line.
stdscr.chgat(0, 21, curses.A_BOLD)
```

The `Textbox` class in the `curses.textpad` module now supports editing in insert mode as well as overwrite mode. Insert mode is enabled by supplying a true value for the *insert_mode* parameter when creating the `Textbox` instance.

- The `datetime` module's `strftime()` methods now support a `%f` format code that expands to the number of microseconds in the object, zero-padded on the left to six places. (Contributed by Skip Montanaro; issue 1158.)

- The `decimal` module was updated to version 1.66 of the General Decimal Specification. New features include some methods for some basic mathematical functions such as `exp()` and `log10()`:

```
>>> Decimal(1).exp()
Decimal("2.718281828459045235360287471")
>>> Decimal("2.7182818").ln()
Decimal("0.9999999895305022877376682436")
>>> Decimal(1000).log10()
Decimal("3")
```

The `as_tuple()` method of `Decimal` objects now returns a named tuple with `sign`, `digits`, and `exponent` fields.

(Implemented by Facundo Batista and Mark Dickinson. Named tuple support added by Raymond Hettinger.)

- The `difflib` module's `SequenceMatcher` class now returns named tuples representing matches, with `a`, `b`, and `size` attributes. (Contributed by Raymond Hettinger.)

- An optional `timeout` parameter, specifying a timeout measured in seconds, was added to the `ftplib.FTP` class constructor as well as the `connect()` method. (Added by Facundo Batista.) Also,

the FTP class's storbinary() and storlines() now take an optional *callback* parameter that will be called with each block of data after the data has been sent. (Contributed by Phil Schwartz; issue 1221598.)

- The reduce() built-in function is also available in the functools module. In Python 3.0, the built-in has been dropped and reduce() is only available from functools; currently there are no plans to drop the built-in in the 2.x series. (Patched by Christian Heimes; issue 1739906.)

- When possible, the getpass module will now use /dev/tty to print a prompt message and read the password, falling back to standard error and standard input. If the password may be echoed to the terminal, a warning is printed before the prompt is displayed. (Contributed by Gregory P. Smith.)

- The glob.glob() function can now return Unicode filenames if a Unicode path was used and Unicode filenames are matched within the directory. (issue 1001604)

- A new function in the heapq module, merge(iter1, iter2, ...), takes any number of iterables returning data in sorted order, and returns a new generator that returns the contents of all the iterators, also in sorted order. For example:

```
>>> list(heapq.merge([1, 3, 5, 9], [2, 8, 16]))
[1, 2, 3, 5, 8, 9, 16]
```

  Another new function, heappushpop(heap, item), pushes *item* onto *heap*, then pops off and returns the smallest item. This is more efficient than making a call to heappush() and then heappop().

  heapq is now implemented to only use less-than comparison, instead of the less-than-or-equal comparison it previously used. This makes heapq's usage of a type match the list.sort() method. (Contributed by Raymond Hettinger.)

- An optional timeout parameter, specifying a timeout measured in seconds, was added to the httplib.HTTPConnection and HTTPSConnection class constructors. (Added by Facundo Batista.)

- Most of the inspect module's functions, such as getmoduleinfo() and getargs(), now return named tuples. In addition to behaving like tuples, the elements of the return value can also be accessed as attributes. (Contributed by Raymond Hettinger.)

  Some new functions in the module include isgenerator(), isgeneratorfunction(), and isabstract().

- The itertools module gained several new functions.

  izip_longest(iter1, iter2, ...[, fillvalue]) makes tuples from each of the elements; if some of the iterables are shorter than others, the missing values are set to *fillvalue*. For example:

```
>>> tuple(itertools.izip_longest([1,2,3], [1,2,3,4,5]))
((1, 1), (2, 2), (3, 3), (None, 4), (None, 5))
```

  product(iter1, iter2, ..., [repeat=N]) returns the Cartesian product of the supplied iterables, a set of tuples containing every possible combination of the elements returned from each iterable.

```
>>> list(itertools.product([1,2,3], [4,5,6]))
[(1, 4), (1, 5), (1, 6),
 (2, 4), (2, 5), (2, 6),
 (3, 4), (3, 5), (3, 6)]
```

  The optional *repeat* keyword argument is used for taking the product of an iterable or a set of iterables with themselves, repeated *N* times. With a single iterable argument, *N*-tuples are returned:

```
>>> list(itertools.product([1,2], repeat=3))
[(1, 1, 1), (1, 1, 2), (1, 2, 1), (1, 2, 2),
 (2, 1, 1), (2, 1, 2), (2, 2, 1), (2, 2, 2)]
```

  With two iterables, *2N*-tuples are returned.

```
>>> list(itertools.product([1,2], [3,4], repeat=2))
[(1, 3, 1, 3), (1, 3, 1, 4), (1, 3, 2, 3), (1, 3, 2, 4),
 (1, 4, 1, 3), (1, 4, 1, 4), (1, 4, 2, 3), (1, 4, 2, 4),
```

```
 (2, 3, 1, 3), (2, 3, 1, 4), (2, 3, 2, 3), (2, 3, 2, 4),
 (2, 4, 1, 3), (2, 4, 1, 4), (2, 4, 2, 3), (2, 4, 2, 4)]
```

`combinations(iterable, r)` returns sub-sequences of length *r* from the elements of *iterable*.

```
>>> list(itertools.combinations('123', 2))
[('1', '2'), ('1', '3'), ('2', '3')]
>>> list(itertools.combinations('123', 3))
[('1', '2', '3')]
>>> list(itertools.combinations('1234', 3))
[('1', '2', '3'), ('1', '2', '4'),
 ('1', '3', '4'), ('2', '3', '4')]
```

`permutations(iter[, r])` returns all the permutations of length *r* of the iterable's elements. If *r* is not specified, it will default to the number of elements produced by the iterable.

```
>>> list(itertools.permutations([1,2,3,4], 2))
[(1, 2), (1, 3), (1, 4),
 (2, 1), (2, 3), (2, 4),
 (3, 1), (3, 2), (3, 4),
 (4, 1), (4, 2), (4, 3)]
```

`itertools.chain(*iterables)` is an existing function in `itertools` that gained a new constructor in Python 2.6. `itertools.chain.from_iterable(iterable)` takes a single iterable that should return other iterables. `chain()` will then return all the elements of the first iterable, then all the elements of the second, and so on.

```
>>> list(itertools.chain.from_iterable([[1,2,3], [4,5,6]]))
[1, 2, 3, 4, 5, 6]
```

(All contributed by Raymond Hettinger.)

- The `logging` module's `FileHandler` class and its subclasses `WatchedFileHandler`, `RotatingFileHandler`, and `TimedRotatingFileHandler` now have an optional *delay* parameter to their constructors. If *delay* is true, opening of the log file is deferred until the first `emit()` call is made. (Contributed by Vinay Sajip.)

  `TimedRotatingFileHandler` also has a *utc* constructor parameter. If the argument is true, UTC time will be used in determining when midnight occurs and in generating filenames; otherwise local time will be used.

- Several new functions were added to the `math` module:

  - `isinf()` and `isnan()` determine whether a given float is a (positive or negative) infinity or a NaN (Not a Number), respectively.
  - `copysign()` copies the sign bit of an IEEE 754 number, returning the absolute value of *x* combined with the sign bit of *y*. For example, `math.copysign(1, -0.0)` returns -1.0. (Contributed by Christian Heimes.)
  - `factorial()` computes the factorial of a number. (Contributed by Raymond Hettinger; issue 2138.)
  - `fsum()` adds up the stream of numbers from an iterable, and is careful to avoid loss of precision through using partial sums. (Contributed by Jean Brouwers, Raymond Hettinger, and Mark Dickinson; issue 2819.)
  - `acosh()`, `asinh()` and `atanh()` compute the inverse hyperbolic functions.
  - `log1p()` returns the natural logarithm of *1+x* (base *e*).
  - `trunc()` rounds a number toward zero, returning the closest `Integral` that's between the function's argument and zero. Added as part of the backport of PEP 3141's type hierarchy for numbers.

- The `math` module has been improved to give more consistent behaviour across platforms, especially with respect to handling of floating-point exceptions and IEEE 754 special values.

  Whenever possible, the module follows the recommendations of the C99 standard about 754's special values. For example, `sqrt(-1.)` should now give a `ValueError` across almost all platforms, while

`sqrt(float('NaN'))` should return a NaN on all IEEE 754 platforms. Where Annex 'F' of the C99 standard recommends signaling 'divide-by-zero' or 'invalid', Python will raise `ValueError`. Where Annex 'F' of the C99 standard recommends signaling 'overflow', Python will raise `OverflowError`. (See issue 711019 and issue 1640.)

(Contributed by Christian Heimes and Mark Dickinson.)

- `mmap` objects now have a `rfind()` method that searches for a substring beginning at the end of the string and searching backwards. The `find()` method also gained an *end* parameter giving an index at which to stop searching. (Contributed by John Lenton.)

- The `operator` module gained a `methodcaller()` function that takes a name and an optional set of arguments, returning a callable that will call the named function on any arguments passed to it. For example:

```
>>> # Equivalent to lambda s: s.replace('old', 'new')
>>> replacer = operator.methodcaller('replace', 'old', 'new')
>>> replacer('old wine in old bottles')
'new wine in new bottles'
```

(Contributed by Georg Brandl, after a suggestion by Gregory Petrosyan.)

The `attrgetter()` function now accepts dotted names and performs the corresponding attribute lookups:

```
>>> inst_name = operator.attrgetter(
...         '__class__.__name__')
>>> inst_name('')
'str'
>>> inst_name(help)
'_Helper'
```

(Contributed by Georg Brandl, after a suggestion by Barry Warsaw.)

- The `os` module now wraps several new system calls. `fchmod(fd, mode)` and `fchown(fd, uid, gid)` change the mode and ownership of an opened file, and `lchmod(path, mode)` changes the mode of a symlink. (Contributed by Georg Brandl and Christian Heimes.)

  `chflags()` and `lchflags()` are wrappers for the corresponding system calls (where they're available), changing the flags set on a file. Constants for the flag values are defined in the `stat` module; some possible values include `UF_IMMUTABLE` to signal the file may not be changed and `UF_APPEND` to indicate that data can only be appended to the file. (Contributed by M. Levinson.)

  `os.closerange(low, high)` efficiently closes all file descriptors from *low* to *high*, ignoring any errors and not including *high* itself. This function is now used by the `subprocess` module to make starting processes faster. (Contributed by Georg Brandl; issue 1663329.)

- The `os.environ` object's `clear()` method will now unset the environment variables using `os.unsetenv()` in addition to clearing the object's keys. (Contributed by Martin Horcicka; issue 1181.)

- The `os.walk()` function now has a `followlinks` parameter. If set to True, it will follow symlinks pointing to directories and visit the directory's contents. For backward compatibility, the parameter's default value is false. Note that the function can fall into an infinite recursion if there's a symlink that points to a parent directory. (issue 1273829)

- In the `os.path` module, the `splitext()` function has been changed to not split on leading period characters. This produces better results when operating on Unix's dot-files. For example, `os.path.splitext('.ipython')` now returns `('.ipython', '')` instead of `('', '.ipython')`. (issue 115886)

  A new function, `os.path.relpath(path, start='.')`, returns a relative path from the `start` path, if it's supplied, or from the current working directory to the destination `path`. (Contributed by Richard Barran; issue 1339796.)

  On Windows, `os.path.expandvars()` will now expand environment variables given in the form "%var%", and "~user" will be expanded into the user's home directory path. (Contributed by Josiah Carlson; issue 957650.)

- The Python debugger provided by the `pdb` module gained a new command: "run" restarts the Python program being debugged and can optionally take new command-line arguments for the program. (Contributed by Rocky Bernstein; issue 1393667.)

- The `pdb.post_mortem()` function, used to begin debugging a traceback, will now use the traceback returned by `sys.exc_info()` if no traceback is supplied. (Contributed by Facundo Batista; issue 1106316.)

- The `pickletools` module now has an `optimize()` function that takes a string containing a pickle and removes some unused opcodes, returning a shorter pickle that contains the same data structure. (Contributed by Raymond Hettinger.)

- A `get_data()` function was added to the `pkgutil` module that returns the contents of resource files included with an installed Python package. For example:

```
>>> import pkgutil
>>> print pkgutil.get_data('test', 'exception_hierarchy.txt')
BaseException
 +-- SystemExit
 +-- KeyboardInterrupt
 +-- GeneratorExit
 +-- Exception
      +-- StopIteration
      +-- StandardError
 ...
```

  (Contributed by Paul Moore; issue 2439.)

- The `pyexpat` module's `Parser` objects now allow setting their `buffer_size` attribute to change the size of the buffer used to hold character data. (Contributed by Achim Gaedke; issue 1137.)

- The `Queue` module now provides queue variants that retrieve entries in different orders. The `PriorityQueue` class stores queued items in a heap and retrieves them in priority order, and `LifoQueue` retrieves the most recently added entries first, meaning that it behaves like a stack. (Contributed by Raymond Hettinger.)

- The `random` module's `Random` objects can now be pickled on a 32-bit system and unpickled on a 64-bit system, and vice versa. Unfortunately, this change also means that Python 2.6's `Random` objects can't be unpickled correctly on earlier versions of Python. (Contributed by Shawn Ligocki; issue 1727780.)

  The new `triangular(low, high, mode)` function returns random numbers following a triangular distribution. The returned values are between *low* and *high*, not including *high* itself, and with *mode* as the most frequently occurring value in the distribution. (Contributed by Wladmir van der Laan and Raymond Hettinger; issue 1681432.)

- Long regular expression searches carried out by the `re` module will check for signals being delivered, so time-consuming searches can now be interrupted. (Contributed by Josh Hoyt and Ralf Schmitt; issue 846388.)

  The regular expression module is implemented by compiling bytecodes for a tiny regex-specific virtual machine. Untrusted code could create malicious strings of bytecode directly and cause crashes, so Python 2.6 includes a verifier for the regex bytecode. (Contributed by Guido van Rossum from work for Google App Engine; issue 3487.)

- The `rlcompleter` module's `Completer.complete()` method will now ignore exceptions triggered while evaluating a name. (Fixed by Lorenz Quack; issue 2250.)

- The `sched` module's `scheduler` instances now have a read-only `queue` attribute that returns the contents of the scheduler's queue, represented as a list of named tuples with the fields `(time, priority, action, argument)`. (Contributed by Raymond Hettinger; issue 1861.)

- The `select` module now has wrapper functions for the Linux `epoll()` and BSD `kqueue()` system calls. `modify()` method was added to the existing `poll` objects; `pollobj.modify(fd, eventmask)` takes a file descriptor or file object and an event mask, modifying the recorded event mask for that file. (Contributed by Christian Heimes; issue 1657.)

- The `shutil.copytree()` function now has an optional *ignore* argument that takes a callable object. This callable will receive each directory path and a list of the directory's contents, and returns a list of names that will be ignored, not copied.

  The `shutil` module also provides an `ignore_patterns()` function for use with this new parameter. `ignore_patterns()` takes an arbitrary number of glob-style patterns and returns a callable that will ignore any files and directories that match any of these patterns. The following example copies a directory tree, but skips both `.svn` directories and Emacs backup files, which have names ending with '~':

  ```
  shutil.copytree('Doc/library', '/tmp/library',
                  ignore=shutil.ignore_patterns('*~', '.svn'))
  ```

  (Contributed by Tarek Ziadé; issue 2663.)

- Integrating signal handling with GUI handling event loops like those used by Tkinter or GTk+ has long been a problem; most software ends up polling, waking up every fraction of a second to check if any GUI events have occurred. The `signal` module can now make this more efficient. Calling `signal.set_wakeup_fd(fd)` sets a file descriptor to be used; when a signal is received, a byte is written to that file descriptor. There's also a C-level function, `PySignal_SetWakeupFd()`, for setting the descriptor.

  Event loops will use this by opening a pipe to create two descriptors, one for reading and one for writing. The writable descriptor will be passed to `set_wakeup_fd()`, and the readable descriptor will be added to the list of descriptors monitored by the event loop via `select()` or `poll()`. On receiving a signal, a byte will be written and the main event loop will be woken up, avoiding the need to poll.

  (Contributed by Adam Olsen; issue 1583.)

  The `siginterrupt()` function is now available from Python code, and allows changing whether signals can interrupt system calls or not. (Contributed by Ralf Schmitt.)

  The `setitimer()` and `getitimer()` functions have also been added (where they're available). `setitimer()` allows setting interval timers that will cause a signal to be delivered to the process after a specified time, measured in wall-clock time, consumed process time, or combined process+system time. (Contributed by Guilherme Polo; issue 2240.)

- The `smtplib` module now supports SMTP over SSL thanks to the addition of the `SMTP_SSL` class. This class supports an interface identical to the existing `SMTP` class. (Contributed by Monty Taylor.) Both class constructors also have an optional `timeout` parameter that specifies a timeout for the initial connection attempt, measured in seconds. (Contributed by Facundo Batista.)

  An implementation of the LMTP protocol (**RFC 2033**) was also added to the module. LMTP is used in place of SMTP when transferring e-mail between agents that don't manage a mail queue. (LMTP implemented by Leif Hedstrom; issue 957003.)

  `SMTP.starttls()` now complies with **RFC 3207** and forgets any knowledge obtained from the server not obtained from the TLS negotiation itself. (Patch contributed by Bill Fenner; issue 829951.)

- The `socket` module now supports TIPC (http://tipc.sf.net), a high-performance non-IP-based protocol designed for use in clustered environments. TIPC addresses are 4- or 5-tuples. (Contributed by Alberto Bertogli; issue 1646.)

  A new function, `create_connection()`, takes an address and connects to it using an optional timeout value, returning the connected socket object. This function also looks up the address's type and connects to it using IPv4 or IPv6 as appropriate. Changing your code to use `create_connection()` instead of `socket(socket.AF_INET, ...)` may be all that's required to make your code work with IPv6.

- The base classes in the `SocketServer` module now support calling a `handle_timeout()` method after a span of inactivity specified by the server's `timeout` attribute. (Contributed by Michael Pomraning.) The `serve_forever()` method now takes an optional poll interval measured in seconds, controlling how often the server will check for a shutdown request. (Contributed by Pedro Werneck and Jeffrey Yasskin; issue 742598, issue 1193577.)

- The `sqlite3` module, maintained by Gerhard Haering, has been updated from version 2.3.2 in Python 2.5 to version 2.4.1.

- The `struct` module now supports the C99 `_Bool` type, using the format character `'?'`. (Contributed by David Remahl.)

- The `Popen` objects provided by the `subprocess` module now have `terminate()`, `kill()`, and `send_signal()` methods. On Windows, `send_signal()` only supports the `SIGTERM` signal, and all these methods are aliases for the Win32 API function `TerminateProcess()`. (Contributed by Christian Heimes.)

- A new variable in the `sys` module, `float_info`, is an object containing information derived from the `float.h` file about the platform's floating-point support. Attributes of this object include `mant_dig` (number of digits in the mantissa), `epsilon` (smallest difference between 1.0 and the next largest value representable), and several others. (Contributed by Christian Heimes; issue 1534.)

  Another new variable, `dont_write_bytecode`, controls whether Python writes any `.pyc` or `.pyo` files on importing a module. If this variable is true, the compiled files are not written. The variable is initially set on start-up by supplying the *-B* switch to the Python interpreter, or by setting the **PYTHONDONTWRITEBYTECODE** environment variable before running the interpreter. Python code can subsequently change the value of this variable to control whether bytecode files are written or not. (Contributed by Neal Norwitz and Georg Brandl.)

  Information about the command-line arguments supplied to the Python interpreter is available by reading attributes of a named tuple available as `sys.flags`. For example, the `verbose` attribute is true if Python was executed in verbose mode, `debug` is true in debugging mode, etc. These attributes are all read-only. (Contributed by Christian Heimes.)

  A new function, `getsizeof()`, takes a Python object and returns the amount of memory used by the object, measured in bytes. Built-in objects return correct results; third-party extensions may not, but can define a `__sizeof__()` method to return the object's size. (Contributed by Robert Schuppenies; issue 2898.)

  It's now possible to determine the current profiler and tracer functions by calling `sys.getprofile()` and `sys.gettrace()`. (Contributed by Georg Brandl; issue 1648.)

- The `tarfile` module now supports POSIX.1-2001 (pax) tarfiles in addition to the POSIX.1-1988 (ustar) and GNU tar formats that were already supported. The default format is GNU tar; specify the `format` parameter to open a file using a different format:

  ```
  tar = tarfile.open("output.tar", "w",
                     format=tarfile.PAX_FORMAT)
  ```

  The new `encoding` and `errors` parameters specify an encoding and an error handling scheme for character conversions. `'strict'`, `'ignore'`, and `'replace'` are the three standard ways Python can handle errors,; `'utf-8'` is a special value that replaces bad characters with their UTF-8 representation. (Character conversions occur because the PAX format supports Unicode filenames, defaulting to UTF-8 encoding.)

  The `TarFile.add()` method now accepts an `exclude` argument that's a function that can be used to exclude certain filenames from an archive. The function must take a filename and return true if the file should be excluded or false if it should be archived. The function is applied to both the name initially passed to `add()` and to the names of files in recursively-added directories.

  (All changes contributed by Lars Gustäbel).

- An optional `timeout` parameter was added to the `telnetlib.Telnet` class constructor, specifying a timeout measured in seconds. (Added by Facundo Batista.)

- The `tempfile.NamedTemporaryFile` class usually deletes the temporary file it created when the file is closed. This behaviour can now be changed by passing `delete=False` to the constructor. (Contributed by Damien Miller; issue 1537850.)

  A new class, `SpooledTemporaryFile`, behaves like a temporary file but stores its data in memory until a maximum size is exceeded. On reaching that limit, the contents will be written to an on-disk temporary file. (Contributed by Dustin J. Mitchell.)

The `NamedTemporaryFile` and `SpooledTemporaryFile` classes both work as context managers, so you can write `with tempfile.NamedTemporaryFile() as tmp: ...`. (Contributed by Alexander Belopolsky; issue 2021.)

- The `test.test_support` module gained a number of context managers useful for writing tests. `EnvironmentVarGuard()` is a context manager that temporarily changes environment variables and automatically restores them to their old values.

  Another context manager, `TransientResource`, can surround calls to resources that may or may not be available; it will catch and ignore a specified list of exceptions. For example, a network test may ignore certain failures when connecting to an external web site:

  ```
  with test_support.TransientResource(IOError,
                                      errno=errno.ETIMEDOUT):
      f = urllib.urlopen('https://sf.net')
      ...
  ```

  Finally, `check_warnings()` resets the `warning` module's warning filters and returns an object that will record all warning messages triggered (issue 3781):

  ```
  with test_support.check_warnings() as wrec:
      warnings.simplefilter("always")
      # ... code that triggers a warning ...
      assert str(wrec.message) == "function is outdated"
      assert len(wrec.warnings) == 1, "Multiple warnings raised"
  ```

  (Contributed by Brett Cannon.)

- The `textwrap` module can now preserve existing whitespace at the beginnings and ends of the newly-created lines by specifying `drop_whitespace=False` as an argument:

  ```
  >>> S = """This  sentence  has a bunch   of
  ...   extra   whitespace."""
  >>> print textwrap.fill(S, width=15)
  This  sentence
  has a bunch
  of    extra
  whitespace.
  >>> print textwrap.fill(S, drop_whitespace=False, width=15)
  This  sentence
    has a bunch
     of    extra
     whitespace.
  >>>
  ```

  (Contributed by Dwayne Bailey; issue 1581073.)

- The `threading` module API is being changed to use properties such as `daemon` instead of `setDaemon()` and `isDaemon()` methods, and some methods have been renamed to use underscores instead of camel-case; for example, the `activeCount()` method is renamed to `active_count()`. Both the 2.6 and 3.0 versions of the module support the same properties and renamed methods, but don't remove the old methods. No date has been set for the deprecation of the old APIs in Python 3.x; the old APIs won't be removed in any 2.x version. (Carried out by several people, most notably Benjamin Peterson.)

  The `threading` module's `Thread` objects gained an `ident` property that returns the thread's identifier, a nonzero integer. (Contributed by Gregory P. Smith; issue 2871.)

- The `timeit` module now accepts callables as well as strings for the statement being timed and for the setup code. Two convenience functions were added for creating `Timer` instances: `repeat(stmt, setup, time, repeat, number)` and `timeit(stmt, setup, time, number)` create an instance and call the corresponding method. (Contributed by Erik Demaine; issue 1533909.)

- The `Tkinter` module now accepts lists and tuples for options, separating the elements by spaces before passing the resulting value to Tcl/Tk. (Contributed by Guilherme Polo; issue 2906.)

- The `turtle` module for turtle graphics was greatly enhanced by Gregor Lingl. New features in the module include:
  - Better animation of turtle movement and rotation.
  - Control over turtle movement using the new `delay()`, `tracer()`, and `speed()` methods.
  - The ability to set new shapes for the turtle, and to define a new coordinate system.
  - Turtles now have an `undo()` method that can roll back actions.
  - Simple support for reacting to input events such as mouse and keyboard activity, making it possible to write simple games.
  - A `turtle.cfg` file can be used to customize the starting appearance of the turtle's screen.
  - The module's docstrings can be replaced by new docstrings that have been translated into another language.

  (issue 1513695)

- An optional `timeout` parameter was added to the `urllib.urlopen()` function and the `urllib.ftpwrapper` class constructor, as well as the `urllib2.urlopen()` function. The parameter specifies a timeout measured in seconds. For example:

```
>>> u = urllib2.urlopen("http://slow.example.com",
                        timeout=3)
Traceback (most recent call last):
  ...
urllib2.URLError: <urlopen error timed out>
>>>
```

  (Added by Facundo Batista.)

- The Unicode database provided by the `unicodedata` module has been updated to version 5.1.0. (Updated by Martin von Loewis; issue 3811.)

- The `warnings` module's `formatwarning()` and `showwarning()` gained an optional *line* argument that can be used to supply the line of source code. (Added as part of issue 1631171, which re-implemented part of the `warnings` module in C code.)

  A new function, `catch_warnings()`, is a context manager intended for testing purposes that lets you temporarily modify the warning filters and then restore their original values (issue 3781).

- The XML-RPC `SimpleXMLRPCServer` and `DocXMLRPCServer` classes can now be prevented from immediately opening and binding to their socket by passing True as the `bind_and_activate` constructor parameter. This can be used to modify the instance's `allow_reuse_address` attribute before calling the `server_bind()` and `server_activate()` methods to open the socket and begin listening for connections. (Contributed by Peter Parente; issue 1599845.)

  `SimpleXMLRPCServer` also has a `_send_traceback_header` attribute; if true, the exception and formatted traceback are returned as HTTP headers "X-Exception" and "X-Traceback". This feature is for debugging purposes only and should not be used on production servers because the tracebacks might reveal passwords or other sensitive information. (Contributed by Alan McIntyre as part of his project for Google's Summer of Code 2007.)

- The `xmlrpclib` module no longer automatically converts `datetime.date` and `datetime.time` to the `xmlrpclib.DateTime` type; the conversion semantics were not necessarily correct for all applications. Code using `xmlrpclib` should convert `date` and `time` instances. (issue 1330538) The code can also handle dates before 1900 (contributed by Ralf Schmitt; issue 2014) and 64-bit integers represented by using `<i8>` in XML-RPC responses (contributed by Riku Lindblad; issue 2985).

- The `zipfile` module's `ZipFile` class now has `extract()` and `extractall()` methods that will unpack a single file or all the files in the archive to the current directory, or to a specified directory:

```
z = zipfile.ZipFile('python-251.zip')

# Unpack a single file, writing it relative
```

```
# to the /tmp directory.
z.extract('Python/sysmodule.c', '/tmp')

# Unpack all the files in the archive.
z.extractall()
```

(Contributed by Alan McIntyre; issue 467924.)

The `open()`, `read()` and `extract()` methods can now take either a filename or a `ZipInfo` object. This is useful when an archive accidentally contains a duplicated filename. (Contributed by Graham Horler; issue 1775025.)

Finally, `zipfile` now supports using Unicode filenames for archived files. (Contributed by Alexey Borzenkov; issue 1734346.)

18.1 The `ast` module

The `ast` module provides an Abstract Syntax Tree representation of Python code, and Armin Ronacher contributed a set of helper functions that perform a variety of common tasks. These will be useful for HTML templating packages, code analyzers, and similar tools that process Python code.

The `parse()` function takes an expression and returns an AST. The `dump()` function outputs a representation of a tree, suitable for debugging:

```
import ast

t = ast.parse("""
d = {}
for i in 'abcdefghijklm':
    d[i + i] = ord(i) - ord('a') + 1
print d
""")
print ast.dump(t)
```

This outputs a deeply nested tree:

```
Module(body=[
  Assign(targets=[
    Name(id='d', ctx=Store())
   ], value=Dict(keys=[], values=[]))
  For(target=Name(id='i', ctx=Store()),
      iter=Str(s='abcdefghijklm'), body=[
    Assign(targets=[
      Subscript(value=
        Name(id='d', ctx=Load()),
          slice=
          Index(value=
            BinOp(left=Name(id='i', ctx=Load()), op=Add(),
             right=Name(id='i', ctx=Load()))), ctx=Store())
     ], value=
     BinOp(left=
      BinOp(left=
       Call(func=
        Name(id='ord', ctx=Load()), args=[
          Name(id='i', ctx=Load())
         ], keywords=[], starargs=None, kwargs=None),
       op=Sub(), right=Call(func=
        Name(id='ord', ctx=Load()), args=[
          Str(s='a')
         ], keywords=[], starargs=None, kwargs=None)),
       op=Add(), right=Num(n=1)))
```

```
    ], orelse=[])
   Print(dest=None, values=[
     Name(id='d', ctx=Load())
   ], nl=True)
 ])
```

The `literal_eval()` method takes a string or an AST representing a literal expression, parses and evaluates it, and returns the resulting value. A literal expression is a Python expression containing only strings, numbers, dictionaries, etc. but no statements or function calls. If you need to evaluate an expression but cannot accept the security risk of using an `eval()` call, `literal_eval()` will handle it safely:

```
>>> literal = '("a", "b", {2:4, 3:8, 1:2})'
>>> print ast.literal_eval(literal)
('a', 'b', {1: 2, 2: 4, 3: 8})
>>> print ast.literal_eval('"a" + "b"')
Traceback (most recent call last):
  ...
ValueError: malformed string
```

The module also includes `NodeVisitor` and `NodeTransformer` classes for traversing and modifying an AST, and functions for common transformations such as changing line numbers.

18.2 The `future_builtins` module

Python 3.0 makes many changes to the repertoire of built-in functions, and most of the changes can't be introduced in the Python 2.x series because they would break compatibility. The `future_builtins` module provides versions of these built-in functions that can be imported when writing 3.0-compatible code.

The functions in this module currently include:

- `ascii(obj)`: equivalent to `repr()`. In Python 3.0, `repr()` will return a Unicode string, while `ascii()` will return a pure ASCII bytestring.
- `filter(predicate, iterable)`, `map(func, iterable1, ...)`: the 3.0 versions return iterators, unlike the 2.x built-ins which return lists.
- `hex(value)`, `oct(value)`: instead of calling the `__hex__()` or `__oct__()` methods, these versions will call the `__index__()` method and convert the result to hexadecimal or octal. `oct()` will use the new `0o` notation for its result.

18.3 The `json` module: JavaScript Object Notation

The new `json` module supports the encoding and decoding of Python types in JSON (Javascript Object Notation). JSON is a lightweight interchange format often used in web applications. For more information about JSON, see http://www.json.org.

`json` comes with support for decoding and encoding most built-in Python types. The following example encodes and decodes a dictionary:

```
>>> import json
>>> data = {"spam" : "foo", "parrot" : 42}
>>> in_json = json.dumps(data) # Encode the data
>>> in_json
'{"parrot": 42, "spam": "foo"}'
>>> json.loads(in_json) # Decode into a Python object
{"spam" : "foo", "parrot" : 42}
```

It's also possible to write your own decoders and encoders to support more types. Pretty-printing of the JSON strings is also supported.

`json` (originally called simplejson) was written by Bob Ippolito.

18.4 The `plistlib` module: A Property-List Parser

The `.plist` format is commonly used on Mac OS X to store basic data types (numbers, strings, lists, and dictionaries) by serializing them into an XML-based format. It resembles the XML-RPC serialization of data types.

Despite being primarily used on Mac OS X, the format has nothing Mac-specific about it and the Python implementation works on any platform that Python supports, so the `plistlib` module has been promoted to the standard library.

Using the module is simple:

```
import sys
import plistlib
import datetime

# Create data structure
data_struct = dict(lastAccessed=datetime.datetime.now(),
                   version=1,
                   categories=('Personal','Shared','Private'))

# Create string containing XML.
plist_str = plistlib.writePlistToString(data_struct)
new_struct = plistlib.readPlistFromString(plist_str)
print data_struct
print new_struct

# Write data structure to a file and read it back.
plistlib.writePlist(data_struct, '/tmp/customizations.plist')
new_struct = plistlib.readPlist('/tmp/customizations.plist')

# read/writePlist accepts file-like objects as well as paths.
plistlib.writePlist(data_struct, sys.stdout)
```

18.5 ctypes Enhancements

Thomas Heller continued to maintain and enhance the `ctypes` module.

`ctypes` now supports a `c_bool` datatype that represents the C99 `bool` type. (Contributed by David Remahl; issue 1649190.)

The `ctypes` string, buffer and array types have improved support for extended slicing syntax, where various combinations of `(start, stop, step)` are supplied. (Implemented by Thomas Wouters.)

All `ctypes` data types now support `from_buffer()` and `from_buffer_copy()` methods that create a ctypes instance based on a provided buffer object. `from_buffer_copy()` copies the contents of the object, while `from_buffer()` will share the same memory area.

A new calling convention tells `ctypes` to clear the `errno` or Win32 LastError variables at the outset of each wrapped call. (Implemented by Thomas Heller; issue 1798.)

You can now retrieve the Unix `errno` variable after a function call. When creating a wrapped function, you can supply `use_errno=True` as a keyword parameter to the `DLL()` function and then call the module-level methods `set_errno()` and `get_errno()` to set and retrieve the error value.

The Win32 LastError variable is similarly supported by the `DLL()`, `OleDLL()`, and `WinDLL()` functions. You supply `use_last_error=True` as a keyword parameter and then call the module-level methods `set_last_error()` and `get_last_error()`.

The `byref()` function, used to retrieve a pointer to a ctypes instance, now has an optional *offset* parameter that is a byte count that will be added to the returned pointer.

18.6 Improved SSL Support

Bill Janssen made extensive improvements to Python 2.6's support for the Secure Sockets Layer by adding a new module, `ssl`, that's built atop the OpenSSL library. This new module provides more control over the protocol negotiated, the X.509 certificates used, and has better support for writing SSL servers (as opposed to clients) in Python. The existing SSL support in the `socket` module hasn't been removed and continues to work, though it will be removed in Python 3.0.

To use the new module, you must first create a TCP connection in the usual way and then pass it to the `ssl.wrap_socket()` function. It's possible to specify whether a certificate is required, and to obtain certificate info by calling the `getpeercert()` method.

See Also:

The documentation for the `ssl` module.

19 Deprecations and Removals

- String exceptions have been removed. Attempting to use them raises a `TypeError`.
- Changes to the `Exception` interface as dictated by **PEP 352** continue to be made. For 2.6, the `message` attribute is being deprecated in favor of the `args` attribute.
- (3.0-warning mode) Python 3.0 will feature a reorganized standard library that will drop many outdated modules and rename others. Python 2.6 running in 3.0-warning mode will warn about these modules when they are imported.

  The list of deprecated modules is: `audiodev`, `bgenlocations`, `buildtools`, `bundlebuilder`, `Canvas`, `compiler`, `dircache`, `dl`, `fpformat`, `gensuitemodule`, `ihooks`, `imageop`, `imgfile`, `linuxaudiodev`, `mhlib`, `mimetools`, `multifile`, `new`, `pure`, `statvfs`, `sunaudiodev`, `test.testall`, and `toaiff`.
- The `gopherlib` module has been removed.
- The `MimeWriter` module and `mimify` module have been deprecated; use the `email` package instead.
- The `md5` module has been deprecated; use the `hashlib` module instead.
- The `posixfile` module has been deprecated; `fcntl.lockf()` provides better locking.
- The `popen2` module has been deprecated; use the `subprocess` module.
- The `rgbimg` module has been removed.
- The `sets` module has been deprecated; it's better to use the built-in `set` and `frozenset` types.
- The `sha` module has been deprecated; use the `hashlib` module instead.

20 Build and C API Changes

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

- Python now must be compiled with C89 compilers (after 19 years!). This means that the Python source tree has dropped its own implementations of `memmove()` and `strerror()`, which are in the C89 standard library.
- Python 2.6 can be built with Microsoft Visual Studio 2008 (version 9.0), and this is the new default compiler. See the `PCbuild` directory for the build files. (Implemented by Christian Heimes.)
- On Mac OS X, Python 2.6 can be compiled as a 4-way universal build. The **configure** script can take a *`--with-universal-archs=[32-bit|64-bit|all]`* switch, controlling whether the binaries are built for 32-bit architectures (x86, PowerPC), 64-bit (x86-64 and PPC-64), or both. (Contributed by Ronald Oussoren.)

- The BerkeleyDB module now has a C API object, available as `bsddb.db.api`. This object can be used by other C extensions that wish to use the `bsddb` module for their own purposes. (Contributed by Duncan Grisby.)

- The new buffer interface, previously described in the PEP 3118 section, adds `PyObject_GetBuffer()` and `PyBuffer_Release()`, as well as a few other functions.

- Python's use of the C stdio library is now thread-safe, or at least as thread-safe as the underlying library is. A long-standing potential bug occurred if one thread closed a file object while another thread was reading from or writing to the object. In 2.6 file objects have a reference count, manipulated by the `PyFile_IncUseCount()` and `PyFile_DecUseCount()` functions. File objects can't be closed unless the reference count is zero. `PyFile_IncUseCount()` should be called while the GIL is still held, before carrying out an I/O operation using the `FILE *` pointer, and `PyFile_DecUseCount()` should be called immediately after the GIL is re-acquired. (Contributed by Antoine Pitrou and Gregory P. Smith.)

- Importing modules simultaneously in two different threads no longer deadlocks; it will now raise an `ImportError`. A new API function, `PyImport_ImportModuleNoBlock()`, will look for a module in `sys.modules` first, then try to import it after acquiring an import lock. If the import lock is held by another thread, an `ImportError` is raised. (Contributed by Christian Heimes.)

- Several functions return information about the platform's floating-point support. `PyFloat_GetMax()` returns the maximum representable floating point value, and `PyFloat_GetMin()` returns the minimum positive value. `PyFloat_GetInfo()` returns an object containing more information from the `float.h` file, such as `"mant_dig"` (number of digits in the mantissa), `"epsilon"` (smallest difference between 1.0 and the next largest value representable), and several others. (Contributed by Christian Heimes; issue 1534.)

- C functions and methods that use `PyComplex_AsCComplex()` will now accept arguments that have a `__complex__()` method. In particular, the functions in the `cmath` module will now accept objects with this method. This is a backport of a Python 3.0 change. (Contributed by Mark Dickinson; issue 1675423.)

- Python's C API now includes two functions for case-insensitive string comparisons, `PyOS_stricmp(char*, char*)` and `PyOS_strnicmp(char*, char*, Py_ssize_t)`. (Contributed by Christian Heimes; issue 1635.)

- Many C extensions define their own little macro for adding integers and strings to the module's dictionary in the `init*` function. Python 2.6 finally defines standard macros for adding values to a module, `PyModule_AddStringMacro` and `PyModule_AddIntMacro()`. (Contributed by Christian Heimes.)

- Some macros were renamed in both 3.0 and 2.6 to make it clearer that they are macros, not functions. `Py_Size()` became `Py_SIZE()`, `Py_Type()` became `Py_TYPE()`, and `Py_Refcnt()` became `Py_REFCNT()`. The mixed-case macros are still available in Python 2.6 for backward compatibility. (issue 1629)

- Distutils now places C extensions it builds in a different directory when running on a debug version of Python. (Contributed by Collin Winter; issue 1530959.)

- Several basic data types, such as integers and strings, maintain internal free lists of objects that can be reused. The data structures for these free lists now follow a naming convention: the variable is always named `free_list`, the counter is always named `numfree`, and a macro `Py<typename>_MAXFREELIST` is always defined.

- A new Makefile target, "make patchcheck", prepares the Python source tree for making a patch: it fixes trailing whitespace in all modified `.py` files, checks whether the documentation has been changed, and reports whether the `Misc/ACKS` and `Misc/NEWS` files have been updated. (Contributed by Brett Cannon.)

  Another new target, "make profile-opt", compiles a Python binary using GCC's profile-guided optimization. It compiles Python with profiling enabled, runs the test suite to obtain a set of profiling results, and then compiles using these results for optimization. (Contributed by Gregory P. Smith.)

20.1 Port-Specific Changes: Windows

- The support for Windows 95, 98, ME and NT4 has been dropped. Python 2.6 requires at least Windows 2000 SP4.
- The new default compiler on Windows is Visual Studio 2008 (version 9.0). The build directories for Visual Studio 2003 (version 7.1) and 2005 (version 8.0) were moved into the PC/ directory. The new `PCbuild` directory supports cross compilation for X64, debug builds and Profile Guided Optimization (PGO). PGO builds are roughly 10% faster than normal builds. (Contributed by Christian Heimes with help from Amaury Forgeot d'Arc and Martin von Loewis.)
- The `msvcrt` module now supports both the normal and wide char variants of the console I/O API. The `getwch()` function reads a keypress and returns a Unicode value, as does the `getwche()` function. The `putwch()` function takes a Unicode character and writes it to the console. (Contributed by Christian Heimes.)
- `os.path.expandvars()` will now expand environment variables in the form "%var%", and "~user" will be expanded into the user's home directory path. (Contributed by Josiah Carlson; issue 957650.)
- The `socket` module's socket objects now have an `ioctl()` method that provides a limited interface to the `WSAIoctl()` system interface.
- The `_winreg` module now has a function, `ExpandEnvironmentStrings()`, that expands environment variable references such as `%NAME%` in an input string. The handle objects provided by this module now support the context protocol, so they can be used in `with` statements. (Contributed by Christian Heimes.)

  `_winreg` also has better support for x64 systems, exposing the `DisableReflectionKey()`, `EnableReflectionKey()`, and `QueryReflectionKey()` functions, which enable and disable registry reflection for 32-bit processes running on 64-bit systems. (issue 1753245)
- The `msilib` module's `Record` object gained `GetInteger()` and `GetString()` methods that return field values as an integer or a string. (Contributed by Floris Bruynooghe; issue 2125.)

20.2 Port-Specific Changes: Mac OS X

- When compiling a framework build of Python, you can now specify the framework name to be used by providing the *`--with-framework-name=`* option to the **configure** script.
- The `macfs` module has been removed. This in turn required the `macostools.touched()` function to be removed because it depended on the `macfs` module. (issue 1490190)
- Many other Mac OS modules have been deprecated and will removed in Python 3.0: `_builtinSuites`, `aepack`, `aetools`, `aetypes`, `applesingle`, `appletrawmain`, `appletrunner`, `argvemulator`, `Audio_mac`, `autoGIL`, `Carbon`, `cfmfile`, `CodeWarrior`, `ColorPicker`, `EasyDialogs`, `Explorer`, `Finder`, `FrameWork`, `findertools`, `ic`, `icglue`, `icopen`, `macerrors`, `MacOS`, `macfs`, `macostools`, `macresource`, `MiniAEFrame`, `Nav`, `Netscape`, `OSATerminology`, `pimp`, `PixMapWrapper`, `StdSuites`, `SystemEvents`, `Terminal`, and `terminalcommand`.

20.3 Port-Specific Changes: IRIX

A number of old IRIX-specific modules were deprecated and will be removed in Python 3.0: `al` and `AL`, `cd`, `cddb`, `cdplayer`, `CL` and `cl`, `DEVICE`, `ERRNO`, `FILE`, `FL` and `fl`, `flp`, `fm`, `GET`, `GLWS`, `GL` and `gl`, `IN`, `IOCTL`, `jpeg`, `panelparser`, `readcd`, `SV` and `sv`, `torgb`, `videoreader`, and `WAIT`.

21 Porting to Python 2.6

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

- Classes that aren't supposed to be hashable should set `__hash__ = None` in their definitions to indicate the fact.
- String exceptions have been removed. Attempting to use them raises a `TypeError`.
- The `__init__()` method of `collections.deque` now clears any existing contents of the deque before adding elements from the iterable. This change makes the behavior match `list.__init__()`.
- `object.__init__()` previously accepted arbitrary arguments and keyword arguments, ignoring them. In Python 2.6, this is no longer allowed and will result in a `TypeError`. This will affect `__init__()` methods that end up calling the corresponding method on `object` (perhaps through using `super()`). See issue 1683368 for discussion.
- The `Decimal` constructor now accepts leading and trailing whitespace when passed a string. Previously it would raise an `InvalidOperation` exception. On the other hand, the `create_decimal()` method of `Context` objects now explicitly disallows extra whitespace, raising a `ConversionSyntax` exception.
- Due to an implementation accident, if you passed a file path to the built-in `__import__()` function, it would actually import the specified file. This was never intended to work, however, and the implementation now explicitly checks for this case and raises an `ImportError`.
- C API: the `PyImport_Import()` and `PyImport_ImportModule()` functions now default to absolute imports, not relative imports. This will affect C extensions that import other modules.
- C API: extension data types that shouldn't be hashable should define their `tp_hash` slot to `PyObject_HashNotImplemented()`.
- The `socket` module exception `socket.error` now inherits from `IOError`. Previously it wasn't a subclass of `StandardError` but now it is, through `IOError`. (Implemented by Gregory P. Smith; issue 1706815.)
- The `xmlrpclib` module no longer automatically converts `datetime.date` and `datetime.time` to the `xmlrpclib.DateTime` type; the conversion semantics were not necessarily correct for all applications. Code using `xmlrpclib` should convert `date` and `time` instances. (issue 1330538)
- (3.0-warning mode) The `Exception` class now warns when accessed using slicing or index access; having `Exception` behave like a tuple is being phased out.
- (3.0-warning mode) inequality comparisons between two dictionaries or two objects that don't implement comparison methods are reported as warnings. `dict1 == dict2` still works, but `dict1 < dict2` is being phased out.

  Comparisons between cells, which are an implementation detail of Python's scoping rules, also cause warnings because such comparisons are forbidden entirely in 3.0.

22 Acknowledgements

The author would like to thank the following people for offering suggestions, corrections and assistance with various drafts of this article: Georg Brandl, Steve Brown, Nick Coghlan, Ralph Corderoy, Jim Jewett, Kent Johnson, Chris Lambacher, Martin Michlmayr, Antoine Pitrou, Brian Warner.

Index