What's New in Python 2.4

Release 1.01

A.M. Kuchling

Contents

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

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

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

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

1 PEP 218: Built-In Set Objects

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

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

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

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

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

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

See Also:

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

2   PEP 237: Unifying Long Integers and Integers

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

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

See Also:

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

3   PEP 289: Generator Expressions

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

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

instead of

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

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

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

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

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

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

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

expressions in this respect.

See Also:

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

4   PEP 292: Simpler String Substitutions

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

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

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

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

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

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

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

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

See Also:

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

5   PEP 318: Decorators for Functions and Methods

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

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

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

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

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

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

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

```
class C:

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

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

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

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

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

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

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

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

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

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

    return wrapper

@require_int
def p1 (arg):
    print arg

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

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

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

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

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

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

See Also:

PEP 318, "*Decorators for Functions, Methods and Classes*"
    Written by Kevin D. Smith, Jim Jewett, and Skip Montanaro. Several people wrote patches implementing function decorators, but the one that was actually checked in was patch #979728, written by Mark Russell.

6 PEP 322: Reverse Iteration

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

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

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

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

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

See Also:

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

7 PEP 324: New subprocess Module

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

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

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

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

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

The constructor has a number of handy options:

- *close_fds* requests that all file descriptors be closed before running the subprocess.

- *cwd* specifies the working directory in which the subprocess will be executed (defaulting to whatever the parent's working directory is).

- *env* is a dictionary specifying environment variables.

- *preexec_fn* is a function that gets called before the child is started.

- *universal_newlines* opens the child's input and output using Python's universal newline feature.

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

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

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

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

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

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

See Also:

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

8   PEP 327: Decimal Data Type

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

8.1   Why is Decimal needed?

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

- The sign, which is positive or negative.

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

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

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

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

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

```
>>> 1.1
1.1000000000000001
```

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

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

Hence, the `Decimal` type was created.

8.2 The `Decimal` type

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

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

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

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

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

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

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

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

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

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

```
>>> a = decimal.Decimal('35.72')
>>> b = decimal.Decimal('1.73')
>>> a+b
Decimal("37.45")
>>> a-b
Decimal("33.99")
>>> a*b
Decimal("61.7956")
>>> a/b
Decimal("20.64739884393063583815028902")
>>> a ** 2
Decimal("1275.9184")
>>> a**b
Traceback (most recent call last):
  ...
decimal.InvalidOperation: x ** (non-integer)
```

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

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

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

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

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

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

8.3 The `Context` type

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

- `prec` is the precision, the number of decimal places.

- `rounding` specifies the rounding mode. The `decimal` module has constants for the various possibilities: `ROUND_DOWN`, `ROUND_CEILING`, `ROUND_HALF_EVEN`, and various others.

- `traps` is a dictionary specifying what happens on encountering certain error conditions: either an exception is raised or a value is returned. Some examples of error conditions are division by zero, loss of precision, and overflow.

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

```
>>> decimal.getcontext().prec
28
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.1428571428571428571428571429")
>>> decimal.getcontext().prec = 9
>>> decimal.Decimal(1) / decimal.Decimal(7)
Decimal("0.142857143")
```

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

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

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

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

See Also:

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

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

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

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

9 PEP 328: Multi-line Imports

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

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

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

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

The PEP also proposes that all `import` statements be absolute imports, with a leading '`.`' character to indicate a relative import. This part of the PEP was not implemented for Python 2.4, but was completed for Python 2.5.

See Also:

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

10 PEP 331: Locale-Independent Float/String Conversions

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

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

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

- `PyOS_ascii_strtod`(*str*, *ptr*) and `PyOS_ascii_atof`(*str*, *ptr*) both convert a string to a C `double`.

- `PyOS_ascii_formatd`(*buffer*, *buf_len*, *format*, *d*) converts a `double` to an ASCII string.

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

See Also:

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

11 Other Language Changes

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

- Decorators for functions and methods were added (PEP 318).

- Built-in `set` and `frozenset` types were added (PEP 218). Other new built-ins include the `reversed(`*seq*`)` function (PEP 322).

- Generator expressions were added (PEP 289).

- Certain numeric expressions no longer return values restricted to 32 or 64 bits (PEP 237).

- You can now put parentheses around the list of names in a `from` *module* `import` *names* statement (PEP 328).

- The `dict.update()` method now accepts the same argument forms as the `dict` constructor. This includes any mapping, any iterable of key/value pairs, and keyword arguments. (Contributed by Raymond Hettinger.)

- The string methods `ljust()`, `rjust()`, and `center()` now take an optional argument for specifying a fill character other than a space. (Contributed by Raymond Hettinger.)

- Strings also gained an `rsplit()` method that works like the `split()` method but splits from the end of the string.

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

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

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

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

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

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

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

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

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

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

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

- There is a new built-in function `sorted(`*iterable*`)` that works like the in-place `list.sort()` method but can be used in expressions. The differences are:

    - the input may be any iterable;

    - a newly formed copy is sorted, leaving the original intact; and

    - the expression returns the new sorted copy

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

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

    (Contributed by Raymond Hettinger.)

- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.

- The interpreter gained a new switch, **-m**, that takes a name, searches for the corresponding module on `sys.path`, and runs the module as a script. For example, you can now run the Python profiler with `python -m profile`. (Contributed by Nick Coghlan.)

- The `eval(`*expr, globals, locals*`)` and `execfile(`*filename, globals, locals*`)` functions and the `exec` statement now accept any mapping type for the *locals* parameter. Previously this had to be a regular Python dictionary. (Contributed by Raymond Hettinger.)

- The `zip()` built-in function and `itertools.izip()` now return an empty list if called with no arguments. Previously they raised a `TypeError` exception. This makes them more suitable for use with variable length argument lists:

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

(Contributed by Raymond Hettinger.)

- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`. The incomplete module object left behind would fool further imports of the same module into succeeding, leading to confusing errors. (Fixed by Tim Peters.)

- `None` is now a constant; code that binds a new value to the name 'None' is now a syntax error. (Contributed by Raymond Hettinger.)

11.1 Optimizations

- The inner loops for list and tuple slicing were optimized and now run about one-third faster. The inner loops for dictionaries were also optimized, resulting in performance boosts for `keys()`, `values()`, `items()`, `iterkeys()`, `itervalues()`, and `iteritems()`. (Contributed by Raymond Hettinger.)

- The machinery for growing and shrinking lists was optimized for speed and for space efficiency. Appending and popping from lists now runs faster due to more efficient code paths and less frequent use of the underlying system `realloc()`. List comprehensions also benefit. `list.extend()` was also optimized and no longer converts its argument into a temporary list before extending the base list. (Contributed by Raymond Hettinger.)

- `list()`, `tuple()`, `map()`, `filter()`, and `zip()` now run several times faster with non-sequence arguments that supply a `__len__()` method. (Contributed by Raymond Hettinger.)

- The methods `list.__getitem__()`, `dict.__getitem__()`, and `dict.__contains__()` are are now implemented as `method_descriptor` objects rather than `wrapper_descriptor` objects. This form of access doubles their performance and makes them more suitable for use as arguments to functionals: '`map(mydict.__getitem__, keylist)`'. (Contributed by Raymond Hettinger.)

- Added a new opcode, `LIST_APPEND`, that simplifies the generated bytecode for list comprehensions and speeds them up by about a third. (Contributed by Raymond Hettinger.)

- The peephole bytecode optimizer has been improved to produce shorter, faster bytecode; remarkably, the resulting bytecode is more readable. (Enhanced by Raymond Hettinger.)

- String concatenations in statements of the form `s = s + "abc"` and `s += "abc"` are now performed more efficiently in certain circumstances. This optimization won't be present in other Python implementations such as Jython, so you shouldn't rely on it; using the `join()` method of strings is still recommended when you want to efficiently glue a large number of strings together. (Contributed by Armin Rigo.)

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

12   New, Improved, and Deprecated Modules

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

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

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

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

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

  - Chinese (PRC): gb2312, gbk, gb18030, big5hkscs, hz

  - Chinese (ROC): big5, cp950

  - Japanese: cp932, euc-jis-2004, euc-jp, euc-jisx0213, iso-2022-jp, iso-2022-jp-1, iso-2022-jp-2, iso-2022-jp-3, iso-2022-jp-ext, iso-2022-jp-2004, shift-jis, shift-jisx0213, shift-jis-2004

  - Korean: cp949, euc-kr, johab, iso-2022-kr

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(Contributed by Hye-Shik Chang.)

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

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

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

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

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

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

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

(Changes implemented by Vinay Sajip.)

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

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

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

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

(Contributed by Raymond Hettinger.)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

    ```
    import threading

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

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

- The `timeit` module now automatically disables periodic garbarge collection during the timing loop. This change makes consecutive timings more comparable. (Contributed by Raymond Hettinger.)

- The `weakref` module now supports a wider variety of objects including Python functions, class instances, sets, frozensets, deques, arrays, files, sockets, and regular expression pattern objects. (Contributed by Raymond Hettinger.)

- The `xmlrpclib` module now supports a multi-call extension for transmitting multiple XML-RPC calls in a single HTTP operation. (Contributed by Brian Quinlan.)

- The `mpz`, `rotor`, and `xreadlines` modules have been removed.

12.1 cookielib

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

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

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

This module was contributed by John J. Lee.

12.2 doctest

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

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

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

finder = doctest.DocTestFinder()

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

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

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

runner.summarize(verbose=1)
```

The above example produces the following output:

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

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

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

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

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

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

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

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

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

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

13 Build and C API Changes

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

- Three new convenience macros were added for common return values from extension functions: `Py_RETURN_NONE`, `Py_RETURN_TRUE`, and `Py_RETURN_FALSE`. (Contributed by Brett Cannon.)

- Another new macro, `Py_CLEAR(obj)`, decreases the reference count of *obj* and sets *obj* to the null pointer. (Contributed by Jim Fulton.)

- A new function, `PyTuple_Pack(N, obj1, obj2, ..., objN)`, constructs tuples from a variable length argument list of Python objects. (Contributed by Raymond Hettinger.)

- A new function, `PyDict_Contains(d, k)`, implements fast dictionary lookups without masking exceptions raised during the look-up process. (Contributed by Raymond Hettinger.)

- The `Py_IS_NAN(X)` macro returns 1 if its float or double argument *X* is a NaN. (Contributed by Tim Peters.)

- C code can avoid unnecessary locking by using the new `PyEval_ThreadsInitialized()` function to tell if any thread operations have been performed. If this function returns false, no lock operations are needed. (Contributed by Nick Coghlan.)

- A new function, `PyArg_VaParseTupleAndKeywords()`, is the same as `PyArg_ParseTupleAndKeywords()` but takes a `va_list` instead of a number of arguments. (Contributed by Greg Chapman.)

- A new method flag, `METH_COEXISTS`, allows a function defined in slots to co-exist with a `PyCFunction` having the same name. This can halve the access time for a method such as `set.__contains__()`. (Contributed by Raymond Hettinger.)

- Python can now be built with additional profiling for the interpreter itself, intended as an aid to people developing the Python core. Providing **--enable-profiling** to the **configure** script will let you profile the interpreter with **gprof**, and providing the **--with-tsc** switch enables profiling using the Pentium's Time-Stamp-Counter register. Note that the **--with-tsc** switch is slightly misnamed, because the profiling feature also works on the PowerPC platform, though that processor architecture doesn't call that register "the TSC register". (Contributed by Jeremy Hylton.)

- The `tracebackobject` type has been renamed to `PyTracebackObject`.

13.1 Port-Specific Changes

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

14 Porting to Python 2.4

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

- Left shifts and hexadecimal/octal constants that are too large no longer trigger a `FutureWarning` and return a value limited to 32 or 64 bits; instead they return a long integer.

- Integer operations will no longer trigger an `OverflowWarning`. The `OverflowWarning` warning will disappear in Python 2.5.

- The `zip()` built-in function and `itertools.izip()` now return an empty list instead of raising a `TypeError` exception if called with no arguments.

- `dircache.listdir()` now passes exceptions to the caller instead of returning empty lists.

- `LexicalHandler.startDTD()` used to receive the public and system IDs in the wrong order. This has been corrected; applications relying on the wrong order need to be fixed.

- `fcntl.ioctl` now warns if the *mutate* argument is omitted and relevant.

- The `tarfile` module now generates GNU-format tar files by default.

- Encountering a failure while importing a module no longer leaves a partially-initialized module object in `sys.modules`.

- `None` is now a constant; code that binds a new value to the name '`None`' is now a syntax error.

15 Acknowledgements