What's New in Python

Release 3.0.1

A. M. Kuchling

February 14, 2009

Python Software Foundation
Email: docs@python.org

Contents

Author Guido van Rossum

Release 3.0.1

Date February 14, 2009

This article explains the new features in Python 3.0, compared to 2.6. Python 3.0, also known as "Python 3000" or "Py3K", is the first ever *intentionally backwards incompatible* Python release. There are more changes than in a typical release, and more that are important for all Python users. Nevertheless, after digesting the changes, you'll find that Python really hasn't changed all that much – by and large, we're mostly fixing well-known annoyances and warts, and removing a lot of old cruft.

This article doesn't attempt to provide a complete specification of all new features, but instead tries to give a convenient overview. For full details, you should refer to the documentation for Python 3.0, and/or the many PEPs referenced in the text. If you want to understand the complete implementation and design rationale for a particular feature, PEPs usually have more details than the regular documentation; but note that PEPs usually are not kept up-to-date once a feature has been fully implemented.

Due to time constraints this document is not as complete as it should have been. As always for a new release, the `Misc/NEWS` file in the source distribution contains a wealth of detailed information about every small thing that was changed.

1 Common Stumbling Blocks

This section lists those few changes that are most likely to trip you up if you're used to Python 2.5.

1.1 Print Is A Function

The `print` statement has been replaced with a `print()` function, with keyword arguments to replace most of the special syntax of the old `print` statement (**PEP 3105**). Examples:

```
Old: print "The answer is", 2*2
New: print("The answer is", 2*2)

Old: print x,           # Trailing comma suppresses newline
New: print(x, end=" ")  # Appends a space instead of a newline

Old: print              # Prints a newline
New: print()            # You must call the function!

Old: print >>sys.stderr, "fatal error"
New: print("fatal error", file=sys.stderr)

Old: print (x, y)       # prints repr((x, y))
New: print((x, y))      # Not the same as print(x, y)!
```

You can also customize the separator between items, e.g.:

```
print("There are <", 2**32, "> possibilities!", sep="")
```

which produces:

```
There are <4294967296> possibilities!
```

Note:

- The `print()` function doesn't support the "softspace" feature of the old `print` statement. For example, in Python 2.x, `print "A\n", "B"` would write `"A\nB\n"`; but in Python 3.0, `print("A\n", "B")` writes `"A\n B\n"`.
- Initially, you'll be finding yourself typing the old `print x` a lot in interactive mode. Time to retrain your fingers to type `print(x)` instead!
- When using the `2to3` source-to-source conversion tool, all `print` statements are automatically converted to `print()` function calls, so this is mostly a non-issue for larger projects.

1.2 Views And Iterators Instead Of Lists

Some well-known APIs no longer return lists:

- `dict` methods `dict.keys()`, `dict.items()` and `dict.values()` return "views" instead of lists. For example, this no longer works: `k = d.keys(); k.sort()`. Use `k = sorted(d)` instead (this works in Python 2.5 too and is just as efficient).
- Also, the `dict.iterkeys()`, `dict.iteritems()` and `dict.itervalues()` methods are no longer supported.
- `map()` and `filter()` return iterators. If you really need a list, a quick fix is e.g. `list(map(...))`, but a better fix is often to use a list comprehension (especially when the original code uses `lambda`), or rewriting the code so it doesn't need a list at all. Particularly tricky is `map()` invoked for the side effects of the function; the correct transformation is to use a regular `for` loop (since creating a list would just be wasteful).
- `range()` now behaves like `xrange()` used to behave, except it works with values of arbitrary size. The latter no longer exists.
- `zip()` now returns an iterator.

1.3 Ordering Comparisons

Python 3.0 has simplified the rules for ordering comparisons:

- The ordering comparison operators (<, <=, >=, >) raise a TypeError exception when the operands don't have a meaningful natural ordering. Thus, expressions like `1 < ''`, `0 > None` or `len <= len` are no longer valid, and e.g. `None < None` raises `TypeError` instead of returning `False`. A corollary is that sorting a heterogeneous list no longer makes sense – all the elements must be comparable to each other. Note that this does not apply to the == and != operators: objects of different incomparable types always compare unequal to each other.
- `builtin.sorted()` and `list.sort()` no longer accept the *cmp* argument providing a comparison function. Use the *key* argument instead. N.B. the *key* and *reverse* arguments are now "keyword-only".
- The `cmp()` function should be treated as gone, and the `__cmp__()` special method is no longer supported. Use `__lt__()` for sorting, `__eq__()` with `__hash__()`, and other rich comparisons as needed. (If you really need the `cmp()` functionality, you could use the expression `(a > b) - (a < b)` as the equivalent for `cmp(a, b)`.)

1.4 Integers

- **PEP 0237**: Essentially, `long` renamed to `int`. That is, there is only one built-in integral type, named `int`; but it behaves mostly like the old `long` type.

- **PEP 0238**: An expression like `1/2` returns a float. Use `1//2` to get the truncating behavior. (The latter syntax has existed for years, at least since Python 2.2.)
- The `sys.maxint` constant was removed, since there is no longer a limit to the value of integers. However, `sys.maxsize` can be used as an integer larger than any practical list or string index. It conforms to the implementation's "natural" integer size and is typically the same as `sys.maxint` in previous releases on the same platform (assuming the same build options).
- The `repr()` of a long integer doesn't include the trailing `L` anymore, so code that unconditionally strips that character will chop off the last digit instead. (Use `str()` instead.)
- Octal literals are no longer of the form `0720`; use `0o720` instead.

1.5 Text Vs. Data Instead Of Unicode Vs. 8-bit

Everything you thought you knew about binary data and Unicode has changed.

- Python 3.0 uses the concepts of *text* and (binary) *data* instead of Unicode strings and 8-bit strings. All text is Unicode; however *encoded* Unicode is represented as binary data. The type used to hold text is `str`, the type used to hold data is `bytes`. The biggest difference with the 2.x situation is that any attempt to mix text and data in Python 3.0 raises `TypeError`, whereas if you were to mix Unicode and 8-bit strings in Python 2.x, it would work if the 8-bit string happened to contain only 7-bit (ASCII) bytes, but you would get `UnicodeDecodeError` if it contained non-ASCII values. This value-specific behavior has caused numerous sad faces over the years.
- As a consequence of this change in philosophy, pretty much all code that uses Unicode, encodings or binary data most likely has to change. The change is for the better, as in the 2.x world there were numerous bugs having to do with mixing encoded and unencoded text. To be prepared in Python 2.x, start using `unicode` for all unencoded text, and `str` for binary or encoded data only. Then the `2to3` tool will do most of the work for you.
- You can no longer use `u"..."` literals for Unicode text. However, you must use `b"..."` literals for binary data.
- As the `str` and `bytes` types cannot be mixed, you must always explicitly convert between them. Use `str.encode()` to go from `str` to `bytes`, and `bytes.decode()` to go from `bytes` to `str`. You can also use `bytes(s, encoding=...)` and `str(b, encoding=...)`, respectively.
- Like `str`, the `bytes` type is immutable. There is a separate *mutable* type to hold buffered binary data, `bytearray`. Nearly all APIs that accept `bytes` also accept `bytearray`. The mutable API is based on `collections.MutableSequence`.
- All backslashes in raw string literals are interpreted literally. This means that `'\U'` and `'\u'` escapes in raw strings are not treated specially. For example, `r'\u20ac'` is a string of 6 characters in Python 3.0, whereas in 2.6, `ur'\u20ac'` was the single "euro" character. (Of course, this change only affects raw string literals; the euro character is `'\u20ac'` in Python 3.0.)
- The builtin `basestring` abstract type was removed. Use `str` instead. The `str` and `bytes` types don't have functionality enough in common to warrant a shared base class. The `2to3` tool (see below) replaces every occurrence of `basestring` with `str`.
- Files opened as text files (still the default mode for `open()`) always use an encoding to map between strings (in memory) and bytes (on disk). Binary files (opened with a `b` in the mode argument) always use bytes in memory. This means that if a file is opened using an incorrect mode or encoding, I/O will likely fail loudly, instead of silently producing incorrect data. It also means that even Unix users will have to specify the correct mode (text or binary) when opening a file. There is a platform-dependent default encoding, which on Unixy platforms can be set with the `LANG` environment variable (and sometimes also with some other platform-specific locale-related environment variables). In many cases, but not all, the system default is UTF-8; you should never count on this

default. Any application reading or writing more than pure ASCII text should probably have a way to override the encoding. There is no longer any need for using the encoding-aware streams in the `codecs` module.

- Filenames are passed to and returned from APIs as (Unicode) strings. This can present platform-specific problems because on some platforms filenames are arbitrary byte strings. (On the other hand, on Windows filenames are natively stored as Unicode.) As a work-around, most APIs (e.g. `open()` and many functions in the `os` module) that take filenames accept `bytes` objects as well as strings, and a few APIs have a way to ask for a `bytes` return value. Thus, `os.listdir()` returns a list of `bytes` instances if the argument is a `bytes` instance, and `os.getcwdb()` returns the current working directory as a `bytes` instance. Note that when `os.listdir()` returns a list of strings, filenames that cannot be decoded properly are omitted rather than raising `UnicodeError`.

- Some system APIs like `os.environ` and `sys.argv` can also present problems when the bytes made available by the system is not interpretable using the default encoding. Setting the `LANG` variable and rerunning the program is probably the best approach.

- **PEP 3138**: The `repr()` of a string no longer escapes non-ASCII characters. It still escapes control characters and code points with non-printable status in the Unicode standard, however.

- **PEP 3120**: The default source encoding is now UTF-8.

- **PEP 3131**: Non-ASCII letters are now allowed in identifiers. (However, the standard library remains ASCII-only with the exception of contributor names in comments.)

- The `StringIO` and `cStringIO` modules are gone. Instead, import the `io` module and use `io.StringIO` or `io.BytesIO` for text and data respectively.

- See also the *Unicode HOWTO* (in), which was updated for Python 3.0.

2 Overview Of Syntax Changes

This section gives a brief overview of every *syntactic* change in Python 3.0.

2.1 New Syntax

- **PEP 3107**: Function argument and return value annotations. This provides a standardized way of annotating a function's parameters and return value. There are no semantics attached to such annotations except that they can be introspected at runtime using the `__annotations__` attribute. The intent is to encourage experimentation through metaclasses, decorators or frameworks.

- **PEP 3102**: Keyword-only arguments. Named parameters occurring after `*args` in the parameter list *must* be specified using keyword syntax in the call. You can also use a bare `*` in the parameter list to indicate that you don't accept a variable-length argument list, but you do have keyword-only arguments.

- Keyword arguments are allowed after the list of base classes in a class definition. This is used by the new convention for specifying a metaclass (see next section), but can be used for other purposes as well, as long as the metaclass supports it.

- **PEP 3104**: `nonlocal` statement. Using `nonlocal x` you can now assign directly to a variable in an outer (but non-global) scope. `nonlocal` is a new reserved word.

- **PEP 3132**: Extended Iterable Unpacking. You can now write things like `a, b, *rest = some_sequence`. And even `*rest, a = stuff`. The `rest` object is always a (possibly empty) list; the right-hand side may be any iterable. Example:

```
(a, *rest, b) = range(5)
```

This sets *a* to `0`, *b* to `4`, and *rest* to `[1, 2, 3]`.

- Dictionary comprehensions: `{k: v for k, v in stuff}` means the same thing as `dict(stuff)` but is more flexible. (This is **PEP 0274** vindicated. :-)
- Set literals, e.g. `{1, 2}`. Note that `{}` is an empty dictionary; use `set()` for an empty set. Set comprehensions are also supported; e.g., `{x for x in stuff}` means the same thing as `set(stuff)` but is more flexible.
- New octal literals, e.g. `0o720` (already in 2.6). The old octal literals (`0720`) are gone.
- New binary literals, e.g. `0b1010` (already in 2.6), and there is a new corresponding builtin function, `bin()`.
- Bytes literals are introduced with a leading `b` or `B`, and there is a new corresponding builtin function, `bytes()`.

2.2 Changed Syntax

- **PEP 3109** and **PEP 3134**: new `raise` statement syntax: '`raise [expr [from expr]]`'. See below.
- `as` and `with` are now reserved words. (Since 2.6, actually.)
- `True`, `False`, and `None` are reserved words. (2.6 partially enforced the restrictions on `None` already.)
- Change from `except` *exc*, *var* to `except` *exc* `as` *var*. See **PEP 3110**.
- **PEP 3115**: New Metaclass Syntax. Instead of:

```
class C:
    __metaclass__ = M
    ...
```

you must now use:

```
class C(metaclass=M):
    ...
```

The module-global `__metaclass__` variable is no longer supported. (It was a crutch to make it easier to default to new-style classes without deriving every class from `object`.)

- List comprehensions no longer support the syntactic form '`[... for var in item1, item2, ...]`'. Use '`[... for var in (item1, item2, ...)]`' instead. Also note that list comprehensions have different semantics: they are closer to syntactic sugar for a generator expression inside a `list()` constructor, and in particular the loop control variables are no longer leaked into the surrounding scope.
- The *ellipsis* (`...`) can be used as an atomic expression anywhere. (Previously it was only allowed in slices.) Also, it *must* now be spelled as `...`. (Previously it could also be spelled as `. . .`, by a mere accident of the grammar.)

2.3 Removed Syntax

- **PEP 3113**: Tuple parameter unpacking removed. You can no longer write `def foo(a, (b, c)): ...`. Use `def foo(a, b_c): b, c = b_c` instead.
- Removed backticks (use `repr()` instead).
- Removed <> (use `!=` instead).
- Removed keyword: `exec()` is no longer a keyword; it remains as a function. (Fortunately the function syntax was also accepted in 2.x.) Also note that `exec()` no longer takes a stream argument; instead of `exec(f)` you can use `exec(f.read())`.

- Integer literals no longer support a trailing `l` or `L`.
- String literals no longer support a leading `u` or `U`.
- The `from` *module* `import *` syntax is only allowed at the module level, no longer inside functions.
- The only acceptable syntax for relative imports is '`from .[module] import name`'. All `import` forms not starting with `.` are interpreted as absolute imports. (**PEP 0328**)
- Classic classes are gone.

3 Changes Already Present In Python 2.6

Since many users presumably make the jump straight from Python 2.5 to Python 3.0, this section reminds the reader of new features that were originally designed for Python 3.0 but that were back-ported to Python 2.6. The corresponding sections in *What's New in Python 2.6* should be consulted for longer descriptions.

- *PEP 343: The 'with' statement*. The `with` statement is now a standard feature and no longer needs to be imported from the `__future__`. Also check out *Writing Context Managers* and *The contextlib module*.
- *PEP 366: Explicit Relative Imports From a Main Module*. This enhances the usefulness of the *-m* option when the referenced module lives in a package.
- *PEP 370: Per-user site-packages Directory*.
- *PEP 371: The multiprocessing Package*.
- *PEP 3101: Advanced String Formatting*. Note: the 2.6 description mentions the `format()` method for both 8-bit and Unicode strings. In 3.0, only the `str` type (text strings with Unicode support) supports this method; the `bytes` type does not. The plan is to eventually make this the only API for string formatting, and to start deprecating the `%` operator in Python 3.1.
- *PEP 3105: print As a Function*. This is now a standard feature and no longer needs to be imported from `__future__`. More details were given above.
- *PEP 3110: Exception-Handling Changes*. The `except` *exc* `as` *var* syntax is now standard and `except` *exc*, *var* is no longer supported. (Of course, the `as` *var* part is still optional.)
- *PEP 3112: Byte Literals*. The `b"..."` string literal notation (and its variants like `b'...'`, `b"""..."""`, and `br"..."`) now produces a literal of type `bytes`.
- *PEP 3116: New I/O Library*. The `io` module is now the standard way of doing file I/O, and the initial values of `sys.stdin`, `sys.stdout` and `sys.stderr` are now instances of `io.TextIOBase`. The builtin `open()` function is now an alias for `io.open()` and has additional keyword arguments *encoding*, *errors*, *newline* and *closefd*. Also note that an invalid *mode* argument now raises `ValueError`, not `IOError`. The binary file object underlying a text file object can be accessed as `f.buffer` (but beware that the text object maintains a buffer of itself in order to speed up the encoding and decoding operations).
- *PEP 3118: Revised Buffer Protocol*. The old builtin `buffer()` is now really gone; the new builtin `memoryview()` provides (mostly) similar functionality.
- *PEP 3119: Abstract Base Classes*. The `abc` module and the ABCs defined in the `collections` module plays a somewhat more prominent role in the language now, and builtin collection types like `dict` and `list` conform to the `collections.MutableMapping` and `collections.MutableSequence` ABCs, respectively.
- *PEP 3127: Integer Literal Support and Syntax*. As mentioned above, the new octal literal notation is the only one supported, and binary literals have been added.

- *PEP 3129: Class Decorators*.
- *PEP 3141: A Type Hierarchy for Numbers*. The `numbers` module is another new use of ABCs, defining Python's "numeric tower". Also note the new `fractions` module which implements `numbers.Rational`.

4 Library Changes

Due to time constraints, this document does not exhaustively cover the very extensive changes to the standard library. **PEP 3108** is the reference for the major changes to the library. Here's a capsule review:

- Many old modules were removed. Some, like `gopherlib` (no longer used) and `md5` (replaced by `hashlib`), were already deprecated by **PEP 0004**. Others were removed as a result of the removal of support for various platforms such as Irix, BeOS and Mac OS 9 (see **PEP 0011**). Some modules were also selected for removal in Python 3.0 due to lack of use or because a better replacement exists. See **PEP 3108** for an exhaustive list.
- The `bsddb3` package was removed because its presence in the core standard library has proved over time to be a particular burden for the core developers due to testing instability and Berkeley DB's release schedule. However, the package is alive and well, externally maintained at http://www.jcea.es/programacion/pybsddb.htm.
- Some modules were renamed because their old name disobeyed **PEP 0008**, or for various other reasons. Here's the list:

Old Name	New Name
_winreg	winreg
ConfigParser	configparser
copy_reg	copyreg
Queue	queue
SocketServer	socketserver
markupbase	_markupbase
repr	reprlib
test.test_support	test.support

- A common pattern in Python 2.x is to have one version of a module implemented in pure Python, with an optional accelerated version implemented as a C extension; for example, `pickle` and `cPickle`. This places the burden of importing the accelerated version and falling back on the pure Python version on each user of these modules. In Python 3.0, the accelerated versions are considered implementation details of the pure Python versions. Users should always import the standard version, which attempts to import the accelerated version and falls back to the pure Python version. The `pickle` / `cPickle` pair received this treatment. The `profile` module is on the list for 3.1. The `StringIO` module has been turned into a class in the `io` module.
- Some related modules have been grouped into packages, and usually the submodule names have been simplified. The resulting new packages are:
  - `dbm` (`anydbm`, `dbhash`, `dbm`, `dumbdbm`, `gdbm`, `whichdb`).
  - `html` (`HTMLParser`, `htmlentitydefs`).
  - `http` (`httplib`, `BaseHTTPServer`, `CGIHTTPServer`, `SimpleHTTPServer`, `Cookie`, `cookielib`).
  - `tkinter` (all `Tkinter`-related modules except `turtle`). The target audience of `turtle` doesn't really care about `tkinter`. Also note that as of Python 2.6, the functionality of `turtle` has been greatly enhanced.
  - `urllib` (`urllib`, `urllib2`, `urlparse`, `robotparse`).
  - `xmlrpc` (`xmlrpclib`, `DocXMLRPCServer`, `SimpleXMLRPCServer`).

Some other changes to standard library modules, not covered by **PEP 3108**:

- Killed `sets`. Use the builtin `set()` function.
- Cleanup of the `sys` module: removed `sys.exitfunc()`, `sys.exc_clear()`, `sys.exc_type`, `sys.exc_value`, `sys.exc_traceback`. (Note that `sys.last_type` etc. remain.)
- Cleanup of the `array.array` type: the `read()` and `write()` methods are gone; use `fromfile()` and `tofile()` instead. Also, the `'c'` typecode for array is gone – use either `'b'` for bytes or `'u'` for Unicode characters.
- Cleanup of the `operator` module: removed `sequenceIncludes()` and `isCallable()`.
- Cleanup of the `thread` module: `acquire_lock()` and `release_lock()` are gone; use `acquire()` and `release()` instead.
- Cleanup of the `random` module: removed the `jumpahead()` API.
- The `new` module is gone.
- The functions `os.tmpnam()`, `os.tempnam()` and `os.tmpfile()` have been removed in favor of the `tempfile` module.
- The `tokenize` module has been changed to work with bytes. The main entry point is now `tokenize.tokenize()`, instead of generate_tokens.
- `string.letters` and its friends (`string.lowercase` and `string.uppercase`) are gone. Use `string.ascii_letters` etc. instead. (The reason for the removal is that `string.letters` and friends had locale-specific behavior, which is a bad idea for such attractively-named global "constants".)
- Renamed module `__builtin__` to `builtins` (removing the underscores, adding an 's'). The `__builtins__` variable found in most global namespaces is unchanged. To modify a builtin, you should use `builtins`, not `__builtins__`!

5 PEP 3101: A New Approach To String Formatting

- A new system for built-in string formatting operations replaces the `%` string formatting operator. (However, the `%` operator is still supported; it will be deprecated in Python 3.1 and removed from the language at some later time.) Read **PEP 3101** for the full scoop.

6 Changes To Exceptions

The APIs for raising and catching exception have been cleaned up and new powerful features added:

- **PEP 0352**: All exceptions must be derived (directly or indirectly) from `BaseException`. This is the root of the exception hierarchy. This is not new as a recommendation, but the *requirement* to inherit from `BaseException` is new. (Python 2.6 still allowed classic classes to be raised, and placed no restriction on what you can catch.) As a consequence, string exceptions are finally truly and utterly dead.
- Almost all exceptions should actually derive from `Exception`; `BaseException` should only be used as a base class for exceptions that should only be handled at the top level, such as `SystemExit` or `KeyboardInterrupt`. The recommended idiom for handling all exceptions except for this latter category is to use `except Exception`.
- `StandardError` was removed (in 2.6 already).
- Exceptions no longer behave as sequences. Use the `args` attribute instead.

- **PEP 3109**: Raising exceptions. You must now use '`raise Exception(args)`' instead of '`raise Exception, args`'. Additionally, you can no longer explicitly specify a traceback; instead, if you *have* to do this, you can assign directly to the `__traceback__` attribute (see below).
- **PEP 3110**: Catching exceptions. You must now use '`except SomeException as variable`' instead of '`except SomeException, variable`'. Moreover, the *variable* is explicitly deleted when the `except` block is left.
- **PEP 3134**: Exception chaining. There are two cases: implicit chaining and explicit chaining. Implicit chaining happens when an exception is raised in an `except` or `finally` handler block. This usually happens due to a bug in the handler block; we call this a *secondary* exception. In this case, the original exception (that was being handled) is saved as the `__context__` attribute of the secondary exception. Explicit chaining is invoked with this syntax:

  ```
  raise SecondaryException() from primary_exception
  ```

  (where *primary_exception* is any expression that produces an exception object, probably an exception that was previously caught). In this case, the primary exception is stored on the `__cause__` attribute of the secondary exception. The traceback printed when an unhandled exception occurs walks the chain of `__cause__` and `__context__` attributes and prints a separate traceback for each component of the chain, with the primary exception at the top. (Java users may recognize this behavior.)
- **PEP 3134**: Exception objects now store their traceback as the `__traceback__` attribute. This means that an exception object now contains all the information pertaining to an exception, and there are fewer reasons to use `sys.exc_info()` (though the latter is not removed).
- A few exception messages are improved when Windows fails to load an extension module. For example, `error code 193` is now `%1 is not a valid Win32 application`. Strings now deal with non-English locales.

7 Miscellaneous Other Changes

7.1 Operators And Special Methods

- `!=` now returns the opposite of `==`, unless `==` returns `NotImplemented`.
- The concept of "unbound methods" has been removed from the language. When referencing a method as a class attribute, you now get a plain function object.
- `__getslice__()`, `__setslice__()` and `__delslice__()` were killed. The syntax `a[i:j]` now translates to `a.__getitem__(slice(i, j))` (or `__setitem__()` or `__delitem__()`, when used as an assignment or deletion target, respectively).
- **PEP 3114**: the standard `next()` method has been renamed to `__next__()`.
- The `__oct__()` and `__hex__()` special methods are removed – `oct()` and `hex()` use `__index__()` now to convert the argument to an integer.
- Removed support for `__members__` and `__methods__`.
- The function attributes named `func_X` have been renamed to use the `__X__` form, freeing up these names in the function attribute namespace for user-defined attributes. To wit, `func_closure`, `func_code`, `func_defaults`, `func_dict`, `func_doc`, `func_globals`, `func_name` were renamed to `__closure__`, `__code__`, `__defaults__`, `__dict__`, `__doc__`, `__globals__`, `__name__`, respectively.
- `__nonzero__()` is now `__bool__()`.

7.2 Builtins

- **PEP 3135**: New `super()`. You can now invoke `super()` without arguments and (assuming this is in a regular instance method defined inside a `class` statement) the right class and instance will automatically be chosen. With arguments, the behavior of `super()` is unchanged.
- **PEP 3111**: `raw_input()` was renamed to `input()`. That is, the new `input()` function reads a line from `sys.stdin` and returns it with the trailing newline stripped. It raises `EOFError` if the input is terminated prematurely. To get the old behavior of `input()`, use `eval(input())`.
- A new builtin `next()` was added to call the `__next__()` method on an object.
- Moved `intern()` to `sys.intern()`.
- Removed: `apply()`. Instead of `apply(f, args)` use `f(*args)`.
- Removed `callable()`. Instead of `callable(f)` you can use `hasattr(f, '__call__')`. The `operator.isCallable()` function is also gone.
- Removed `coerce()`. This function no longer serves a purpose now that classic classes are gone.
- Removed `execfile()`. Instead of `execfile(fn)` use `exec(open(fn).read())`.
- Removed `file`. Use `open()`.
- Removed `reduce()`. Use `functools.reduce()` if you really need it; however, 99 percent of the time an explicit `for` loop is more readable.
- Removed `reload()`. Use `imp.reload()`.
- Removed. `dict.has_key()` – use the `in` operator instead.

8 Build and C API Changes

Due to time constraints, here is a *very* incomplete list of changes to the C API.

- Support for several platforms was dropped, including but not limited to Mac OS 9, BeOS, RISCOS, Irix, and Tru64.
- **PEP 3118**: New Buffer API.
- **PEP 3121**: Extension Module Initialization & Finalization.
- **PEP 3123**: Making `PyObject_HEAD` conform to standard C.
- No more C API support for restricted execution.
- `PyNumber_Coerce`, `PyNumber_CoerceEx`, `PyMember_Get`, and `PyMember_Set` C APIs are removed.
- New C API `PyImport_ImportModuleNoBlock`, works like `PyImport_ImportModule` but won't block on the import lock (returning an error instead).
- Renamed the boolean conversion C-level slot and method: `nb_nonzero` is now `nb_bool`.
- Removed `METH_OLDARGS` and `WITH_CYCLE_GC` from the C API.

9 Performance

The net result of the 3.0 generalizations is that Python 3.0 runs the pystone benchmark around 10% slower than Python 2.5. Most likely the biggest cause is the removal of special-casing for small integers. There's room for improvement, but it will happen after 3.0 is released!

10 Porting To Python 3.0

For porting existing Python 2.5 or 2.6 source code to Python 3.0, the best strategy is the following:

1. (Prerequisite:) Start with excellent test coverage.
2. Port to Python 2.6. This should be no more work than the average port from Python 2.x to Python 2.(x+1). Make sure all your tests pass.
3. (Still using 2.6:) Turn on the `-3` command line switch. This enables warnings about features that will be removed (or change) in 3.0. Run your test suite again, and fix code that you get warnings about until there are no warnings left, and all your tests still pass.
4. Run the `2to3` source-to-source translator over your source code tree. (See *2to3 - Automated Python 2 to 3 code translation* (in *The Python Library Reference*) for more on this tool.) Run the result of the translation under Python 3.0. Manually fix up any remaining issues, fixing problems until all tests pass again.

It is not recommended to try to write source code that runs unchanged under both Python 2.6 and 3.0; you'd have to use a very contorted coding style, e.g. avoiding `print` statements, metaclasses, and much more. If you are maintaining a library that needs to support both Python 2.6 and Python 3.0, the best approach is to modify step 3 above by editing the 2.6 version of the source code and running the `2to3` translator again, rather than editing the 3.0 version of the source code.

For porting C extensions to Python 3.0, please see *Porting Extension Modules to 3.0* (in).

Index

P