What's New In Python 3.0

(editors: check NEWS.help for information about editing NEWS using ReST.)

What's New in Python 3.0 final

Release date: 03-Dec-2008

Core and Builtins

- Issue #3996: On Windows, the PyOS_CheckStack function would cause the interpreter to abort ("Fatal Python error: Could not reset the stack!") instead of throwing a MemoryError.
- Issue #3689: The list reversed iterator now supports __length_hint__ instead of __len__. Behavior now matches other reversed iterators.
- Issue #4367: Python would segfault during compiling when the unicodedata module couldn't be imported and N escapes were present.
- Fix build failure of _cursesmodule.c building with -D_FORTIFY_SOURCE=2.

Library

- Issue #4387: binascii now refuses to accept str as binary input.
- Issue #4073: Add 2to3 support to build_scripts, refactor that support in build_py.
- IDLE would print a "Unhandled server exception!" message when internal debugging is enabled.
- Issue #4455: IDLE failed to display the windows list when two windows have the same title.
- Issue #3741: DISTUTILS_USE_SDK set causes msvc9compiler.py to raise an exception.
- Issue #4433: Fixed an access violation when garbage collecting _ctypes.COMError instances.
- Issue #4429: Fixed UnicodeDecodeError in ctypes.
- Issue #4373: Corrected a potential reference leak in the pickle module and silenced a false positive ref leak in distutils.tests.test_build_ext.

- Issue #4382: dbm.dumb did not specify the expected file encoding for opened files.
- Issue #4383: When IDLE cannot make the connection to its subprocess, it would fail to properly display the error message.

Build

- Issue #4407: Fix source file that caused the compileall step in Windows installer to fail.

Docs

- Issue #4449: Fixed multiprocessing examples
- Issue #3799: Document that dbm.gnu and dbm.ndbm will accept string arguments for keys and values which will be converted to bytes before committal.

What's New in Python 3.0 release candidate 3?

Release date: 20-Nov-2008

Core and Builtins

- Issue #4349: sys.path included a non-existent platform directory because of a faulty Makefile.
- Issue #3327: Don't overallocate in the modules_by_index list.
- Issue #1721812: Binary set operations and copy() returned the input type instead of the appropriate base type. This was incorrect because set subclasses would be created without their __init__() method being called. The corrected behavior brings sets into line with lists and dicts.
- Issue #4296: Fix PyObject_RichCompareBool so that "x in [x]" evaluates to True, even when x doesn't compare equal to itself. This was a regression from 2.6.
- Issue #3705: Command-line arguments were not correctly decoded when the terminal does not use UTF8.

Library

- Issue #4363: The uuid.uuid1() and uuid.uuid4() functions now work even if the ctypes module is not present.
- FileIO's mode attribute now always includes `"b"`.
- Issue #3799: Fix dbm.dumb to accept strings as well as bytes for keys. String keys are now written out in UTF-8.
- Issue #4338: Fix distutils upload command.
- Issue #4354: Fix distutils register command.
- Issue #4116: Resolve member name conflict in ScrolledCanvas.__init__.
- Issue #4307: The named tuple that `inspect.getfullargspec()` returns now uses `kwonlydefaults` instead of

`kwdefaults`.

- Issue #4298: Fix a segfault when pickle.loads is passed a ill-formed input.
- Issue #4283: Fix a left-over "iteritems" call in distutils.

Build

- Issue #4389: Add icon to the uninstall entry in "add-and-remove-programs".
- Issue #4289: Remove Cancel button from AdvancedDlg.
- Issue #1656675: Register a drop handler for .py* files on Windows.

Tools/Demos

- Demos of the socketserver module now work with Python 3.

What's New in Python 3.0 release candidate 2

Release date: 05-Nov-2008

Core and Builtins

- Issue #4211: The __path__ attribute of frozen packages is now a list instead of a string as required by PEP 302.
- Issue #3727: Fixed poplib.
- Issue #3714: Fixed nntplib by using bytes where appropriate.
- Issue #1210: Fixed imaplib and its documentation.
- Issue #4233: Changed semantic of `_fileio.FileIO`'s `close()` method on file objects with closefd=False. The file descriptor is still kept open but the file object behaves like a closed file. The `FileIO` object also got a new readonly attribute `closefd`.
- Issue #3626: On cygwin, starting python with a non-existent script name would not display anything if the file name is only 1 character long.
- Issue #4176: Fixed a crash when pickling an object which `__reduce__` method does not return iterators for the 4th and 5th items.
- Issue #3723: Fixed initialization of subinterpreters.
- Issue #4213: The file system encoding is now normalized by the codec subsystem, for example UTF-8 is turned into utf-8.
- Issue #4200: Changed the atexit module to store its state in its PyModuleDef atexitmodule. This fixes a bug with multiple subinterpeters.
- Issue #4237: io.FileIO() was raising invalid warnings caused by insufficient initialization of PyFileIOObject struct members.
- Issue #4170: Pickling a collections.defaultdict object would crash the interpreter.
- Issue #4146: Compilation on OpenBSD has been restored.

- Issue #3574: compile() incorrectly handled source code encoded as Latin-1.
- Issues #2384 and #3975: Tracebacks were not correctly printed when the source file contains a `coding:` header: the wrong line was displayed, and the encoding was not respected.
- Issue #3740: Null-initialize module state.
- Issue #3946: PyObject_CheckReadBuffer crashed on a memoryview object.
- Issue #1688: On Windows, the input() prompt was not correctly displayed if it contains non-ascii characters.
- Bug #3951: Py_USING_MEMORY_DEBUGGER should not be enabled by default.

Library

- Issue #3664: The pickle module could segfault if a subclass of Pickler fails to call the base __init__ method.
- Issue #3725: telnetlib now works completely in bytes.
- Issue #4072: Restore build_py_2to3.
- Issue #4014: Don't claim that Python has an Alpha release status, in addition to claiming it is Mature.
- Issue #3187: Add sys.setfilesystemencoding.
- Issue #3187: Better support for "undecodable" filenames. Code by Victor Stinner, with small tweaks by GvR.
- Issue #3965: Allow repeated calls to turtle.Screen, by making it a true singleton object.
- Issue #3911: ftplib.FTP.makeport() could give invalid port numbers.
- Issue #3929: When the database cannot be opened, dbm.open() would incorrectly raise a TypeError: "'tuple' object is not callable" instead of the expected dbm.error.
- Bug #3884: Make the turtle module toplevel again.
- Issue #3547: Fixed ctypes structures bitfields of varying integer sizes.

Extension Modules

- Issue #3659: Subclasses of str didn't work as SQL parameters.

Build

- Issue #4120: Exclude manifest from extension modules in VS2008.
- Issue #4091: Install pythonxy.dll in system32 again.
- Issue #4018: Disable "for me" installations on Vista.
- Issue #4204: Fixed module build errors on FreeBSD 4.

Tools/Demos

- Issue #3717: Fix Demo/embed/demo.c.
- Issue #4072: Add a distutils demo for build_py_2to3.

What's New in Python 3.0 release candidate 1

Release date: 17-Sep-2008

Core and Builtins

- Issue #3827: memoryview lost its size attribute in favor of using len(view).
- Issue #3813: could not lanch python.exe via symbolic link on cygwin.
- Issue #3705: fix crash when given a non-ascii value on the command line for the "-c" and "-m" parameters. Now the behaviour is as expected under Linux, although under Windows it fails at a later point.
- Issue #3279: Importing site at interpreter was failing silently because the site module uses the open builtin which was not initialized at the time.
- Issue #3660: Corrected a reference leak in str.encode() when the encoder does not return a bytes object.
- Issue #3774: Added a few more checks in PyTokenizer_FindEncoding to handle error conditions.
- Issue #3594: Fix Parser/tokenizer.c:fp_setreadl() to open the file being tokenized by either a file path or file pointer for the benefit of PyTokenizer_FindEncoding().
- Issue #3696: Error parsing arguments on OpenBSD <= 4.4 and Cygwin. On these systems, the mbstowcs() function is slightly buggy and must be replaced with strlen() for the purpose of counting of number of wide characters needed to represent the multi-byte character string.
- Issue #3697: "Fatal Python error: Cannot recover from stack overflow" could be easily encountered under Windows in debug mode when exercising the recursion limit checking code, due to bogus handling of recursion limit when USE_STACKCHEK was enabled.
- Issue 3639: The _warnings module could segfault the interpreter when unexpected types were passed in as arguments.
- Issue #3712: The memoryview object had a reference leak and didn't support cyclic garbage collection.
- Issue #3668: Fix a memory leak with the "s*" argument parser in PyArg_ParseTuple and friends, which occurred when the argument for "s*" was correctly parsed but parsing of subsequent arguments failed.
- Issue #3611: An exception __context__ could be cleared in a complex pattern involving a __del__ method re-raising an exception.
- Issue #2534: speed up isinstance() and issubclass() by 50-70%, so as to match Python 2.5 speed despite the __instancecheck__ / __subclasscheck__ mechanism. In the process, fix a bug where isinstance() and issubclass(), when given a tuple of classes as second argument, were looking up __instancecheck__ / __subclasscheck__ on the tuple rather than on each type object.
- Issue #3663: Py_None was decref'd when printing SyntaxErrors.
- Issue #3651: Fix various memory leaks when using the buffer interface, or when the "s#" code of PyArg_ParseTuple is given a bytes object.

- Issue #3657: Fix uninitialized memory read when pickling longs. Found by valgrind.
- Apply security patches from Apple.
- Fix crashes on memory allocation failure found with failmalloc.
- Fix memory leaks found with valgrind and update suppressions file.
- Fix compiler warnings in opt mode which would lead to invalid memory reads.
- Fix problem using wrong name in decimal module reported by pychecker.
- Issue #3650: Fixed a reference leak in bytes.split('x').
- bytes(o) now tries to use o.__bytes__() before using fallbacks.
- Issue #1204: The configure script now tests for additional libraries that may be required when linking against readline. This fixes issues with x86_64 builds on some platforms (a few Linux flavors and OpenBSD).

C API

- PyObject_Bytes and PyBytes_FromObject were added.

Library

- Issue #3756: make re.escape() handle bytes as well as str.
- Issue #3800: fix filter() related bug in formatter.py.
- Issue #874900: fix behaviour of threading module after a fork.
- Issue #3535: zipfile couldn't read some zip files larger than 2GB.
- Issue #3776: Deprecate the bsddb package for removal in 3.0.
- Issue #3762: platform.architecture() fails if python is lanched via its symbolic link.
- Issue #3660: fix a memory leak in the C accelerator of the pickle module.
- Issue #3160: the "bdist_wininst" distutils command didn't work.
- Issue #1658: tkinter changes dict size during iteration in both tkinter.BaseWidget and tkinter.scrolledtext.ScrolledText.
- The bsddb module (and therefore the dbm.bsd module) has been removed. It is now maintained outside of the standard library at http://www.jcea.es/programacion/pybsddb.htm (http://www.jcea.es/programacion/pybsddb.htm).
- Issue 600362: Relocated parse_qs() and parse_qsl(), from the cgi module to the urlparse one. Added a DeprecationWarning in the old module, it will be deprecated in the future.
- Issue #3719: platform.architecture() fails if there are spaces in the path to the Python binary.
- Issue 3602: As part of the merge of r66135, make the parameters on warnings.catch_warnings() keyword-only. Also remove a DeprecationWarning.
- The deprecation warnings for the camelCase threading API names were removed.
- Issue #3110: multiprocessing fails to compiel on solaris 10 due to missing SEM_VALUE_MAX.

Extension Modules

- Issue #3782: os.write() must not accept unicode strings.
- Issue #2975: When compiling several extension modules with Visual Studio 2008 from the same python interpreter, some environment variables would grow without limit.
- Issue #3643: Added a few more checks to _testcapi to prevent segfaults by exploitation of poor argument checking.
- bsddb code updated to version 4.7.3pre2. This code is the same than Python 2.6 one, since the intention is to keep an unified 2.x/3.x codebase. The Python code is automatically translated using "2to3". Please, do not update this code in Python 3.0 by hand. Update the 2.6 one and then do "2to3".
- The _bytesio and _stringio modules are now compiled into the python binary.
- Issue #3492 and #3790: Fixed the zlib module and zipimport module uses of mutable bytearray objects where they should have been using immutable bytes.
- Issue #3797: Fixed the dbm, marshal, mmap, ossaudiodev, & winreg modules to return bytes objects instead of bytearray objects.

Tools/Demos

- Fix Misc/gdbinit so it works.

Build

- Issue #3812: Failed to build python if configure --without-threads.
- Issue #3791: Remove the bsddb module from the Windows installer, and the core bsddb library from the Windows build files.

What's new in Python 3.0b3?

Release date: 20-Aug-2008

Core and Builtins

- Issue #3653: Fix a segfault when sys.excepthook was called with invalid arguments.

- Issue #2394: implement more of the memoryview API, with the caveat that only one-dimensional contiguous buffers are supported and exercised right now. Slicing, slice assignment and comparison (equality and inequality) have been added. Also, the tolist() method has been implemented, but only for byte buffers. Endly, the API has been updated to return bytes objects wherever it used to return bytearrays.

- Issue #3560: clean up the new C PyMemoryView API so that naming is internally consistent; add macros PyMemoryView_GET_BASE() and PyMemoryView_GET_BUFFER() to access useful properties of a memory views without relying on a particular implementation; remove the ill-named PyMemoryView() function (PyMemoryView_GET_BUFFER() can be used instead).

- ctypes function pointers that are COM methods have a boolean True value again.

- Issue #1819: function calls with several named parameters are now on average 35% faster (as measured by pybench).

- The undocumented C APIs PyUnicode_AsString() and PyUnicode_AsStringAndSize() were made private to the interpreter, in order to be able to refine their interfaces for Python 3.1.

  If you need to access the UTF-8 representation of a Unicode object as bytes string, please use PyUnicode_AsUTF8String() instead.

- Issue #3460: PyUnicode_Join() implementation is 10% to 80% faster thanks to Python 3.0's stricter semantics which allow to avoid successive reallocations of the result string (this also affects str.join()).

Library

- Issue #1276: Added temporary aliases for CJK Mac encodings to resolve a build problem on MacOS with CJK locales. It adds four temporary mappings to existing legacy codecs that are virtually compatible with Mac encodings. They will be replaced by codecs correctly implemented in 3.1.
- Issue #3614: Corrected a typo in xmlrpc.client, leading to a NameError "global name 'header' is not defined".
- Issue #2834: update the regular expression library to match the unicode standards of py3k. In other words, mixing bytes and unicode strings (be it as pattern, search string or replacement string) raises a TypeError. Moreover, the re.UNICODE flag is enabled automatically for unicode patterns, and can be disabled by specifying a new re.ASCII flag; as for bytes patterns, ASCII matching is the only option and trying to specify re.UNICODE for such patterns raises a ValueError.
- Issue #3300: make urllib.parse.[un]quote() default to UTF-8. Code contributed by Matt Giuca. quote() now encodes the input before quoting, unquote() decodes after unquoting. There are new arguments to change the encoding and errors settings. There are also new APIs to skip the encode/decode steps. [un]quote_plus() are also affected.
- Issue #2235: numbers.Number now blocks inheritance of the default id() based hash because that hash mechanism is not correct for numeric types. All concrete numeric types that inherit from Number (rather than just registering with it) must explicitly provide a hash implementation in order for their instances to be hashable.
- Issue #2676: in the email package, content-type parsing was hanging on pathological input because of quadratic or exponential behaviour of a regular expression.
- Issue #3476: binary buffered reading through the new "io" library is now thread-safe.
- Issue #1342811: Fix leak in Tkinter.Menu.delete. Commands associated to menu entries were not deleted.
- Remove the TarFileCompat class from tarfile.py.

- Issue #2491: os.fdopen is now almost an alias for the built-in open(), and accepts the same parameters. It just checks that its first argument is an integer.
- Issue #3394: zipfile.writestr sets external attributes when passed a file name rather than a ZipInfo instance, so files are extracted with mode 0600 rather than 000 under Unix.
- Issue #2523: Fix quadratic behaviour when read()ing a binary file without asking for a specific length.

Extension Modules

- Bug #3542: Support Unicode strings in _msi module.

What's new in Python 3.0b2?

Release date: 17-Jul-2008

Core and Builtins

- Issue #3008: the float type has a new instance method 'float.hex' and a new class method 'float.fromhex' to convert floating-point numbers to and from hexadecimal strings, respectively.
- Issue #3083: Add alternate (#) formatting for bin, oct, hex output for str.format(). This adds the prefix 0b, 0o, or 0x, respectively.
- Issue #3280: like chr(), the "%c" format now accepts unicode code points beyond the Basic Multilingual Plane (above 0xffff) on all configurations. On "narrow Unicode" builds, the result is a string of 2 code units, forming a UTF-16 surrogate pair.
- Issue #3282: str.isprintable() should return False for undefined Unicode characters.
- Issue #3236: Return small longs from PyLong_FromString.
- Exception tracebacks now support exception chaining.

Library

- Removed the sunaudio module. Use sunau instead.
- Issue #3554: ctypes.string_at and ctypes.wstring_at did call Python api functions without holding the GIL, which could lead to a fatal error when they failed.
- Issue #799428: Fix Tkinter.Misc._nametowidget to unwrap Tcl command objects.
- Removed "ast" function aliases from the parser module.
- Issue #3313: Fixed a crash when a failed dlopen() call does not set a valid dlerror() message.
- Issue #3258: Fixed a crash when a ctypes POINTER type to an incomplete structure was created.
- Issue #2683: Fix inconsistency in subprocess.Popen.communicate(): the argument now must be a bytes object in any case.
- Issue #3145: help("modules whatever") failed when trying to load the source code of every single module of the standard library, including invalid files used in the test suite.

- The gettext library now consistently uses Unicode strings for message ids and message strings, and `ugettext()` and the like don't exist anymore.
- The traceback module has been expanded to handle chained exceptions.

C API

- Issue #3247: the function Py_FindMethod was removed. Modern types should use the tp_methods slot instead.

Tools/Demos

- The Mac/Demos directory has been removed.
- All of the Mac scripts have been removed (including BuildApplet.py).

What's new in Python 3.0b1?

Release date: 18-Jun-2008

Core and Builtins

- Issue #3211: warnings.warn_explicit() did not guard against its 'registry' argument being anything other than a dict or None. Also fixed a bug in error handling when 'message' and 'category' were both set to None, triggering a bus error.
- Issue #3100: Corrected a crash on deallocation of a subclassed weakref which holds the last (strong) reference to its referent.
- Issue #2630: implement PEP 3138. repr() now returns printable Unicode characters unescaped, to get an ASCII-only representation of an object use ascii().
- Issue #1342: On windows, Python could not start when installed in a directory with non-ascii characters.
- Implement PEP 3121: new module initialization and finalization API.
- Removed the already-defunct `-t` option.
- Issue #2957: Corrected a ValueError "recursion limit exceeded", when unmarshalling many code objects, which happens when importing a large .pyc file (~1000 functions).
- Issue #2963: fix merging oversight that disabled method cache for all types.
- Issue #2964: fix a missing INCREF in instancemethod_descr_get.
- Issue #2895: Don't crash when given bytes objects as keyword names.
- Issue #2798: When parsing arguments with PyArg_ParseTuple, the "s" code now allows any unicode string and returns a utf-8 encoded buffer, just like the "s#" code already does. The "z" code was corrected as well.
- Issue #2863: generators now have a `gen.__name__` attribute that equals `gen.gi_code.co_name`, like `func.__name___` that equals `func.func_code.co_name`. The repr() of a generator now also contains this name.
- Issue #2831: enumerate() now has a `start` argument.

- Issue #2801: fix bug in the float.is_integer method where a ValueError was sometimes incorrectly raised.
- The `--with-toolbox-glue` option (and the associated pymactoolbox.h) have been removed.
- Issue #2196: hasattr() now lets exceptions which do not inherit Exception (KeyboardInterrupt, and SystemExit) propagate instead of ignoring them.
- #3021 Exception reraising sematics have been significantly improved. However, f_exc_type, f_exc_value, and f_exc_traceback cannot be accessed from Python code anymore.
- Three of PyNumberMethods' members, nb_coerce, nb_hex, and nb_oct, have been removed.

Extension Modules

- Renamed `_winreg` module to `winreg`.
- Support os.O_ASYNC and fcntl.FASYNC if the constants exist on the platform.
- Support for Windows 9x has been removed from the winsound module.
- Issue #2870: cmathmodule.c compile error.

Library

- The methods `is_in_tuple()`, `is_vararg()`, and `is_keywordarg()` of symtable.Symbol have been removed.
- Patch #3133: http.server.CGIHTTPRequestHandler did not work on windows.
- a new `urllib` package was created. It consists of code from `urllib`, `urllib2`, `urlparse`, and `robotparser`. The old modules have all been removed. The new package has five submodules: `urllib.parse`, `urllib.request`, `urllib.response`, `urllib.error`, and `urllib.robotparser`. The `urllib.request.urlopen()` function uses the url opener from `urllib2`. (Note that the unittests have not been renamed for the beta, but they will be renamed in the future.)
- rfc822 has been removed in favor of the email package.
- mimetools has been removed in favor of the email package.
- Patch #2849: Remove use of rfc822 module from standard library.
- Added C optimized implementation of io.StringIO.
- The `pickle` module is now automatically use an optimized C implementation of Pickler and Unpickler when available. The `cPickle` module is no longer needed.
- Removed the `htmllib` and `sgmllib` modules.
- The deprecated `SmartCookie` and `SimpleCookie` classes have been removed from `http.cookies`.
- The `commands` module has been removed. Its getoutput() and getstatusoutput() functions have been moved to the `subprocess` module.
- The `http` package was created; it contains the old `httplib` as `http.client`, `Cookie` as `http.cookies`, `cookielib` as

`http.cookiejar`, and the content of the three `HTTPServer` modules as `http.server`.

- The `xmlrpc` package was created; it contains the old `xmlrpclib` module as `xmlrpc.client` and the content of the old `SimpleXMLRPCServer` and `DocXMLRPCServer` modules as `xmlrpc.server`.
- The `dbm` package was created, containing the old modules `anydbm` and `whichdb` in its `__init__.py`, and having `dbm.gnu` (was `gdbm`), `dbm.bsd` (was `dbhash`), `dbm.ndbm` (was `dbm`) and `dbm.dumb` (was `dumbdbm`) as submodules.
- The `repr` module has been renamed to `reprlib`.
- The `statvfs` module has been removed.
- Issue #1713041: fix pprint's handling of maximum depth.
- Issue #2250: Exceptions raised during evaluation of names in rlcompleter's `Completer.complete()` method are now caught and ignored.
- Patch #2659: Added `break_on_hyphens` option to textwrap's `TextWrapper` class.
- Issue #2487: change the semantics of math.ldexp(x, n) when n is too large to fit in a C long. ldexp(x, n) now returns a zero (with suitable sign) if n is large and negative; previously, it raised OverflowError.
- The `ConfigParser` module has been renamed to `configparser`.
- Issue #2865: webbrowser.open() works again in a KDE environment.
- The `multifile` module has been removed.
- The `SocketServer` module has been renamed to `socketserver`.
- Fixed the `__all__` setting on `collections` to include `UserList` and `UserString`.
- The sre module has been removed.
- The Queue module has been renamed to queue.
- The copy_reg module has been renamed to copyreg.
- The mhlib module has been removed.
- The ihooks module has been removed.
- The fpformat module has been removed.
- The dircache module has been removed.
- The Canvas module has been removed.
- The Decimal module gained the magic methods __round__, __ceil__, __floor__ and __trunc__, to give support for round, math.ceil, math.floor and math.trunc.
- The user module has been removed.
- The mutex module has been removed.
- The imputil module has been removed.
- os.path.walk has been removed in favor of os.walk.
- pdb gained the "until" command.
- The test.test_support module has been renamed to test.support.

- The threading module API was renamed to be PEP 8 compliant. The old names are still present, but will be removed in the near future.

Tools/Demos

- The bgen tool has been removed.

Build

What's New in Python 3.0a5?

Release date: 08-May-2008

Core and Builtins

- Fixed misbehaviour of PyLong_FromSsize_t on systems where sizeof(size_t) > sizeof(long).
- Issue #2221: Corrected a SystemError "error return without exception set", when the code executed by exec() raises an exception, and sys.stdout.flush() also raises an error.
- Bug #2565: The repr() of type objects now calls them 'class', not 'type' - whether they are builtin types or not.
- The command line processing was converted to pass Unicode strings through as unmodified as possible; as a consequence, the C API related to command line arguments was changed to use wchar_t.
- All backslashes in raw strings are interpreted literally. This means that 'u' and 'U' escapes are not treated specially.

Extension Modules

Library

- ctypes objects now support the PEP3118 buffer interface.
- Issue #2682: ctypes callback functions now longer contain a cyclic reference to themselves.
- Issue #2058: Remove the buf attribute and add __slots__ to the TarInfo class in order to reduce tarfile's memory usage.
- Bug #2606: Avoid calling .sort() on a dict_keys object.
- The bundled libffi copy is now in sync with the recently released libffi3.0.5 version, apart from some small changes to Modules/_ctypes/libffi/configure.ac.

Build

- Issue #1496032: On alpha, use -mieee when gcc is the compiler.

- "make install" is now an alias for "make altinstall", to prevent accidentally overwriting a Python 2.x installation. Use "make fullinstall" to force Python 3.0 to be installed as "python".
- Issue #2544: On HP-UX systems, use 'gcc -shared' for linking when gcc is used as compiler.

What's New in Python 3.0a4?

Release date: 02-Apr-2008

Core and Builtins

- Bug #2301: Don't try decoding the source code into the original encoding for syntax errors.

Extension Modules

- The dl module was removed, use the ctypes module instead.
- Use wchar_t functions in _locale module.

Library

- The class distutils.commands.build_py.build_py_2to3 can be used as a build_py replacement to automatically run 2to3 on modules that are going to be installed.
- A new pickle protocol (protocol 3) is added with explicit support for bytes. This is the default protocol. It intentionally cannot be unpickled by Python 2.x.
- When a pickle written by Python 2.x contains an (8-bit) str instance, this is now decoded to a (Unicode) str instance. The encoding used to do this defaults to ASCII, but can be overridden via two new keyword arguments to the Unpickler class. Previously this would create bytes instances, which is usually wrong: str instances are often used to pickle attribute names etc., and text is more common than binary data anyway.
- Default to ASCII as the locale.getpreferredencoding, if the POSIX system doesn't support CODESET and LANG isn't set or doesn't allow deduction of an encoding.
- Issue #1202: zlib.crc32 and zlib.adler32 now return an unsigned value.
- Issue #719888: Updated tokenize to use a bytes API. generate_tokens has been renamed tokenize and now works with bytes rather than strings. A new detect_encoding function has been added for determining source file encoding according to PEP-0263. Token sequences returned by tokenize always start with an ENCODING token which specifies the encoding used to decode the file. This token is used to encode the output of untokenize back to bytes.

What's New in Python 3.0a3?

Release date: 29-Feb-2008

Core and Builtins

- Issue #2282: io.TextIOWrapper was not overriding seekable() from io.IOBase.
- Issue #2115: Important speedup in setting __slot__ attributes. Also prevent a possible crash: an Abstract Base Class would try to access a slot on a registered virtual subclass.
- Fixed repr() and str() of complex numbers with infinity or nan as real or imaginary part.
- Clear all free list during a gc.collect() of the highest generation in order to allow pymalloc to free more arenas. Python may give back memory to the OS earlier.
- Issue #2045: Fix an infinite recursion triggered when printing a subclass of collections.defaultdict, if its default_factory is set to a bound method.
- Fixed a minor memory leak in dictobject.c. The content of the free list was not freed on interpreter shutdown.
- Limit free list of method and builtin function objects to 256 entries each.
- Patch #1953: Added `sys._compact_freelists()` and the C API functions `PyInt_CompactFreeList` and `PyFloat_CompactFreeList` to compact the internal free lists of pre-allocted ints and floats.
- Bug #1983: Fixed return type of fork(), fork1() and forkpty() calls. Python expected the return type int but the fork familie returns pi_t.
- Issue #1678380: Fix a bug that identifies 0j and -0j when they appear in the same code unit.
- Issue #2025: Added tuple.count() and tuple.index() methods to comply with the collections.Sequence API.
- Fixed multiple reinitialization of the Python interpreter. The small int list in longobject.c has caused a seg fault during the third finalization.
- Issue #1973: bytes.fromhex('') raised SystemError.
- Issue #1771: remove cmp parameter from sorted() and list.sort().
- Issue #1969: split and rsplit in bytearray are inconsistent.
- map() no longer accepts None for the first argument. Use zip() instead.
- Issue #1769: Now int("- 1") is not allowed any more.
- Object/longobject.c: long(float('nan')) raises an OverflowError instead of returning 0.
- Issue #1762972: __file__ points to the source file instead of the pyc/pyo file if the py file exists.
- Issue #1393: object_richcompare() returns NotImplemented instead of False if the objects aren't equal, to give the other side a chance.
- Issue #1692: Interpreter was not displaying location of SyntaxError.
- Improve some exception messages when Windows fails to load an extension module. Now we get for example '%1 is not a valid Win32 application' instead of 'error code 193'. Also use Unicode strings to deal with non-English locales.
- Issue #1587: Added instancemethod wrapper for PyCFunctions. The Python C API has gained a new type *PyInstanceMethod_Type* and the functions *PyInstanceMethod_Check(o)*, *PyInstanceMethod_New(func)* and *PyInstanceMethod_Function(im)*.
- Constants gc.DEBUG_OBJECT and gc.DEBUG_INSTANCE have been removed from the gc module; gc.DEBUG_COLLECTABLE or

gc.DEBUG_UNCOLLECTABLE are now enough to print the corresponding list of objects considered by the garbage collector.

- Issue #1573: Improper use of the keyword-only syntax makes the parser crash.
- Issue #1564: The set implementation should special-case PyUnicode instead of PyString.
- Patch #1031213: Decode source line in SyntaxErrors back to its original source encoding.
- inspect.getsource() includes the decorators again.
- Bug #1713: posixpath.ismount() claims symlink to a mountpoint is a mountpoint.
- Fix utf-8-sig incremental decoder, which didn't recognise a BOM when the first chunk fed to the decoder started with a BOM, but was longer than 3 bytes.

Extension Modules

- Code for itertools ifilter(), imap(), and izip() moved to bultins and renamed to filter(), map(), and zip(). Also, renamed izip_longest() to zip_longest() and ifilterfalse() to filterfalse().
- Issue #1762972: Readded the reload() function as imp.reload().
- Bug #2111: mmap segfaults when trying to write a block opened with PROT_READ.
- Issue #2063: correct order of utime and stime in os.times() result on Windows.

Library

- Weakref dictionaries now inherit from MutableMapping.
- Created new UserDict class in collections module. This one inherits from and complies with the MutableMapping ABC. Also, moved UserString and UserList to the collections module. The MutableUserString class was removed.
- Removed UserDict.DictMixin. Replaced all its uses with collections.MutableMapping.
- Issue #1703: getpass() should flush after writing prompt.
- Issue #1585: IDLE uses non-existent xrange() function.
- Issue #1578: Problems in win_getpass.

Build

- Renamed --enable-unicode configure flag to --with-wide-unicode, since Unicode strings can't be disabled anymore.

C API

- Issue #1629: Renamed Py_Size, Py_Type and Py_Refcnt to Py_SIZE, Py_TYPE and Py_REFCNT.
- New API PyImport_ImportModuleNoBlock(), works like PyImport_ImportModule() but won't block on the import lock (returning an error instead).

What's New in Python 3.0a2?

Release date: 07-Dec-2007

(Note: this list is incomplete.)

Core and Builtins

- str8 now has the same construction signature as bytes.
- Comparisons between str and str8 now return False/True for ==/!=. sqlite3 returns str8 when recreating on object from it's __conform__ value. The struct module returns str8 for all string-related formats. This was true before this change, but becomes more apparent thanks to string comparisons always being False.
- Replaced *PyFile_FromFile()* with *PyFile_FromFd(fd, name. mode, buffer, encoding, newline)*.
- Fixed *imp.find_module()* to obey the -- *coding:* -- header.
- Changed *__file__* and *co_filename* to unicode. The path names are decoded with *Py_FileSystemDefaultEncoding* and a new API method *PyUnicode_DecodeFSDefault(char*)* was added.
- io.open() and _fileio.FileIO have grown a new argument closefd. A false value disables the closing of the file descriptor.
- Added a new option -b to issues warnings (-bb for errors) about certain operations between bytes/buffer and str like str(b'') and comparison.
- The standards streams sys.stdin, stdout and stderr may be None when the when the C runtime library returns an invalid file descriptor for the streams (fileno(stdin) < 0). For now this happens only for Windows GUI apps and scripts started with *pythonw.exe*.
- Added PCbuild9 directory for VS 2008.
- Renamed structmember.h WRITE_RESTRICTED to PY_WRITE_RESTRICTED to work around a name clash with VS 2008 on Windows.
- Unbound methods are gone for good. ClassObject.method returns an ordinary function object, instance.method still returns a bound method object. The API of bound methods is cleaned up, too. The im_class attribute is removed and im_func + im_self are renamed to __func__ and __self__. The factory PyMethod_New takes only func and instance as argument.
- intobject.h is no longer included by Python.h. The remains were moved to longobject.h. It still exists to define several aliases from PyInt to PyLong functions.
- Removed sys.maxint, use sys.maxsize instead.

Extension Modules

- The *hotshot* profiler has been removed; use *cProfile* instead.

Library

- When loading an external file using testfile(), the passed-in encoding argument was being ignored if __loader__ is defined and forcing the source to be UTF-8.
- The methods *os.tmpnam()*, *os.tempnam()* and *os.tmpfile()* have been removed in favor of the tempfile module.
- Removed the 'new' module.
- Removed all types from the 'types' module that are easily accessable through builtins.

What's New in Python 3.0a1?

Release date: 31-Aug-2007

Core and Builtins

- PEP 3131: Support non-ASCII identifiers.
- PEP 3120: Change default encoding to UTF-8.
- PEP 3123: Use proper C inheritance for PyObject.
- Removed the __oct__ and __hex__ special methods and added a bin() builtin function.
- PEP 3127: octal literals now start with "0o". Old-style octal literals are invalid. There are binary literals with a prefix of "0b". This also affects int(x, 0).
- None, True, False are now keywords.
- PEP 3119: isinstance() and issubclass() can be overridden.
- Remove BaseException.message.
- Remove tuple parameter unpacking (PEP 3113).
- Remove the f_restricted attribute from frames. This naturally leads to the removal of PyEval_GetRestricted() and PyFrame_IsRestricted().
- PEP 3132 was accepted. That means that you can do `a, *b = range(5)` to assign 0 to a and [1, 2, 3, 4] to b.
- range() now returns an iterator rather than a list. Floats are not allowed. xrange() is no longer defined.
- Patch #1660500: hide iteration variable in list comps, add set comps and use common code to handle compilation of iterative expressions.

- By default, != returns the opposite of ==, unless the latter returns NotImplemented.
- Patch #1680961: sys.exitfunc has been removed and replaced with a private C-level API.
- PEP 3115: new metaclasses: the metaclass is now specified as a keyword arg in the class statement, which can now use the full syntax of a parameter list. Also, the metaclass can implement a __prepare__ function which will be called to create the dictionary for the new class namespace.
- The long-deprecated argument "pend" of PyFloat_FromString() has been removed.
- The dir() function has been extended to call the __dir__() method on its argument, if it exists. If not, it will work like before. This allows customizing the output of dir() in the presence of a __getattr__().
- Removed support for __members__ and __methods__.
- Removed indexing/slicing on BaseException.
- input() became raw_input(): the name input() now implements the functionality formerly known as raw_input(); the name raw_input() is no longer defined.
- Classes listed in an 'except' clause must inherit from BaseException.
- PEP 3106: dict.iterkeys(), .iteritems(), .itervalues() are now gone; and .keys(), .items(), .values() return dict views, which behave like sets.
- PEP 3105: print is now a function. Also (not in the PEP) the 'softspace' attribute of files is now gone (since print() doesn't use it). A side effect of this change is that you can get incomplete output lines in interactive sessions:

  ```
  >>> print(42, end="")
  42>>>
  ```

  We may be able to fix this after the I/O library rewrite.
- PEP 3102: keyword-only arguments.
- Int/Long unification is complete. The 'long' built-in type and literals with trailing 'L' or 'l' have been removed. Performance may be sub-optimal (haven't really benchmarked).
- 'except E, V' must now be spelled as 'except E as V' and deletes V at the end of the except clause; V must be a simple name.

- Added function annotations per PEP 3107.

- Added nonlocal declaration from PEP 3104:

```
>>> def f(x):
...     def inc():
...         nonlocal x
...         x += 1
...         return x
...     return inc
...
>>> inc = f(0)
>>> inc()
1
>>> inc()
2
```

- Moved intern() to sys.intern().

- exec is now a function.

- Renamed nb_nonzero to nb_bool and __nonzero__ to __bool__.

- Classic classes are a thing of the past. All classes are new style.

- Exceptions *must* derive from BaseException.

- Integer division always returns a float. The -Q option is no more. All the following are gone:

  - PyNumber_Divide and PyNumber_InPlaceDivide
  - __div__, __rdiv__, and __idiv__
  - nb_divide, nb_inplace_divide
  - operator.div, operator.idiv, operator.__div__, operator.__idiv__ (Only __truediv__ and __floordiv__ remain, not sure how to handle them if we want to re-use __div__ and friends. If we do, it will make it harder to write code for both 2.x and 3.x.)
- 'as' and 'with' are keywords.

- Absolute import is the default behavior for 'import foo' etc.

- Removed support for syntax: backticks (ie, *x*), <>.
- Removed these Python builtins: apply(), callable(), coerce(), execfile(), file(), reduce(), reload().
- Removed these Python methods: {}.has_key.
- Removed these opcodes: BINARY_DIVIDE, INPLACE_DIVIDE, UNARY_CONVERT.
- Remove C API support for restricted execution.
- zip(), map() and filter() now return iterators, behaving like their itertools counterparts. This also affect map()'s behavior on sequences of unequal length -- it now stops after the shortest one is exhausted.
- Additions: set literals, set comprehensions, ellipsis literal.
- Added class decorators per PEP 3129.

Extension Modules

- Removed the imageop module. Obsolete long with its unit tests becoming useless from the removal of rgbimg and imgfile.
- Removed these attributes from the operator module: div, idiv, __div__, __idiv__, isCallable, sequenceIncludes.
- Removed these attributes from the sys module: exc_clear(), exc_type, exc_value, exc_traceback.

Library

- Removed the compiler package. Use of the _ast module and (an eventual) AST -> bytecode mechanism.
- Removed these modules: audiodev, Bastion, bsddb185, exceptions, linuxaudiodev, md5, MimeWriter, mimify, popen2, rexec, sets, sha, stringold, strop, sunaudiodev, timing, xmllib.
- Moved the toaiff module to Tools/Demos.
- Removed obsolete IRIX modules: al/AL, cd/CD, cddb, cdplayer, cl/CL, DEVICE, ERRNO, FILE, fl/FL, flp, fm, GET, gl/GL, GLWS, IN, imgfile, IOCTL, jpeg, panel, panelparser, readcd, sgi, sv/SV, torgb, WAIT.
- Removed obsolete functions: commands.getstatus(), os.popen*().
- Removed functions in the string module that are also string methods; Remove string.{letters, lowercase, uppercase}.
- Removed support for long obsolete platforms: plat-aix3, plat-irix5.
- Removed xmlrpclib.SlowParser. It was based on xmllib.
- Patch #1680961: atexit has been reimplemented in C.
- Add new codecs for UTF-32, UTF-32-LE and UTF-32-BE.

Build

C API

- Removed these Python slots: __coerce__, __div__, __idiv__, __rdiv__.
- Removed these C APIs: PyNumber_Coerce(), PyNumber_CoerceEx(), PyMember_Get, PyMember_Set.
- Removed these C slots/fields: nb_divide, nb_inplace_divide.
- Removed these macros: staticforward, statichere, PyArg_GetInt, PyArg_NoArgs, _PyObject_Del.
- Removed these typedefs: intargfunc, intintargfunc, intobjargproc, intintobjargproc, getreadbufferproc, getwritebufferproc, getsegcountproc, getcharbufferproc, memberlist.

Tests

- Removed test.testall as test.regrtest replaces it.

Documentation

Mac

- The cfmfile module was removed.

Platforms

- Support for BeOS and AtheOS was removed (according to PEP 11).
- Support for RiscOS, Irix, Tru64 was removed (alledgedly).

Tools/Demos

(For information about older versions, consult the HISTORY file.)

Tweets Follow

Python Software @ThePSF 7 May
PSF Sponsors SciPy 2014 goo.gl/fb/X1vRK
Expand

Python Software @ThePSF 7 May
We're happy to be sponsoring @SciPyConf in Austin, TX! pyfound.blogspot.com/2014/05/psf-sp…
Expand

Python Software @ThePSF 13 Apr
Congratulations to @raymondh, recipient of the PSF's Distinguished Service Award at #pycon2014
Expand

Brian Curtin @brian_curtin 12 Apr
$11,679 raised at the #pyladies auction after @ThePSF pitched in an extra $1,500. This is awesome.
Retweeted by Python Software
Expand

Python Software @ThePSF 10 Apr
PSF Python Brochure now available! Get your copy in Montreal! goo.gl/fb/9BKNJ

Tweet to @ThePSF

The PSF

The Python Software Foundation is the organization behind Python. Become a member of the PSF and help advance the software and our mission.

Back to Top

Back to Top

What's New In Python 3.1

Author: Raymond Hettinger
Release: 3.1.5
Date: April 09, 2012

This article explains the new features in Python 3.1, compared to 3.0.

PEP 372: Ordered Dictionaries

Regular Python dictionaries iterate over key/value pairs in arbitrary order. Over the years, a number of authors have written alternative implementations that remember the order that the keys were originally inserted. Based on the experiences from those implementations, a new `collections.OrderedDict` class has been introduced.

The OrderedDict API is substantially the same as regular dictionaries but will iterate over keys and values in a guaranteed order depending on when a key was first inserted. If a new entry overwrites an existing entry, the original insertion position is left unchanged. Deleting an entry and reinserting it will move it to the end.

The standard library now supports use of ordered dictionaries in several modules. The `configparser` module uses them by default. This lets configuration files be read, modified, and then written back in their original order. The *_asdict()* method for `collections.namedtuple()` now returns an ordered dictionary with the values appearing in the same order as the underlying tuple indicies. The `json` module is being built-out with an *object_pairs_hook* to allow OrderedDicts to be built by the decoder. Support was also added for third-party tools like PyYAML.

See also:

PEP 372 - Ordered Dictionaries
PEP written by Armin Ronacher and Raymond Hettinger. Implementation written by Raymond Hettinger.

Since an ordered dictionary remembers its insertion order, it can be used in conjuction with sorting to make a sorted dictionary:

```
>>> # regular unsorted dictionary
>>> d = {'banana': 3, 'apple':4, 'pear': 1, 'orange': 2}

>>> # dictionary sorted by key
>>> OrderedDict(sorted(d.items(), key=lambda t: t[0]))
```

```
OrderedDict([('apple', 4), ('banana', 3), ('orange', 2), ('pear', 1)])

>>> # dictionary sorted by value
>>> OrderedDict(sorted(d.items(), key=lambda t: t[1]))
OrderedDict([('pear', 1), ('orange', 2), ('banana', 3), ('apple', 4)])

>>> # dictionary sorted by length of the key string
>>> OrderedDict(sorted(d.items(), key=lambda t: len(t[0])))
OrderedDict([('pear', 1), ('apple', 4), ('orange', 2), ('banana', 3)])
```

The new sorted dictionaries maintain their sort order when entries are deleted. But when new keys are added, the keys are appended to the end and the sort is not maintained.

PEP 378: Format Specifier for Thousands Separator

The builtin `format()` function and the `str.format()` method use a mini-language that now includes a simple, non-locale aware way to format a number with a thousands separator. That provides a way to humanize a program's output, improving its professional appearance and readability:

```
>>> format(1234567, ',d')
'1,234,567'
>>> format(1234567.89, ',.2f')
'1,234,567.89'
>>> format(12345.6 + 8901234.12j, ',f')
'12,345.600000+8,901,234.120000j'
>>> format(Decimal('1234567.89'), ',f')
'1,234,567.89'
```

The supported types are `int`, `float`, `complex` and `decimal.Decimal`.

Discussions are underway about how to specify alternative separators like dots, spaces, apostrophes, or underscores. Locale-aware applications should use the existing *n* format specifier which already has some support for thousands separators.

See also:

PEP 378 - Format Specifier for Thousands Separator
PEP written by Raymond Hettinger and implemented by Eric Smith and Mark Dickinson.

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/zipfile is automatically inserted as the first entry in sys.path. (Suggestion and initial patch by Andy Chu; revised patch by Phillip J. Eby and Nick Coghlan; issue 1739468.)

- The `int()` type gained a `bit_length` method that returns the number of bits necessary to represent its argument in binary:

  ```
  >>> n = 37
  >>> bin(37)
  '0b100101'
  >>> n.bit_length()
  6
  >>> n = 2**123-1
  >>> n.bit_length()
  123
  >>> (n+1).bit_length()
  124
  ```

  (Contributed by Fredrik Johansson, Victor Stinner, Raymond Hettinger, and Mark Dickinson; issue 3439.)

- The fields in `format()` strings can now be automatically numbered:

  ```
  >>> 'Sir {} of {}'.format('Gallahad', 'Camelot')
  'Sir Gallahad of Camelot'
  ```

  Formerly, the string would have required numbered fields such as: `'Sir {0} of {1}'`.

  (Contributed by Eric Smith; issue 5237.)

- The `string.maketrans()` function is deprecated and is replaced by new static methods, `bytes.maketrans()` and `bytearray.maketrans()`. This change solves the confusion around which types were supported by the `string` module. Now, `str`, `bytes`, and `bytearray` each have their own **maketrans** and **translate** methods with intermediate translation tables of the appropriate type.

  (Contributed by Georg Brandl; issue 5675.)

- The syntax of the `with` statement now allows multiple context managers in a single statement:

  ```
  >>> with open('mylog.txt') as infile, open('a.out', 'w') as outfile:
  ...     for line in infile:
  ...         if '<critical>' in line:
  ...             outfile.write(line)
  ```

With the new syntax, the `contextlib.nested()` function is no longer needed and is now deprecated.

(Contributed by Georg Brandl and Mattias Brändström; appspot issue 53094.)

- `round(x, n)` now returns an integer if *x* is an integer. Previously it returned a float:

  ```
  >>> round(1123, -2)
  1100
  ```

  (Contributed by Mark Dickinson; issue 4707.)

- Python now uses David Gay's algorithm for finding the shortest floating point representation that doesn't change its value. This should help mitigate some of the confusion surrounding binary floating point numbers.

  The significance is easily seen with a number like `1.1` which does not have an exact equivalent in binary floating point. Since there is no exact equivalent, an expression like `float('1.1')` evaluates to the nearest representable value which is `0x1.199999999999ap+0` in hex or `1.100000000000000088817841970012523233890533447265625` in decimal. That nearest value was and still is used in subsequent floating point calculations.

  What is new is how the number gets displayed. Formerly, Python used a simple approach. The value of `repr(1.1)` was computed as `format(1.1, '.17g')` which evaluated to `'1.1000000000000001'`. The advantage of using 17 digits was that it relied on IEEE-754 guarantees to assure that `eval(repr(1.1))` would round-trip exactly to its original value. The disadvantage is that many people found the output to be confusing (mistaking intrinsic limitations of binary floating point representation as being a problem with Python itself).

  The new algorithm for `repr(1.1)` is smarter and returns `'1.1'`. Effectively, it searches all equivalent string representations (ones that get stored with the same underlying float value) and returns the shortest representation.

  The new algorithm tends to emit cleaner representations when possible, but it does not change the underlying values. So, it is still the case that `1.1 + 2.2 != 3.3` even though the representations may suggest otherwise.

  The new algorithm depends on certain features in the underlying floating point implementation. If the required features are not found, the old algorithm will continue to be used. Also, the text pickle protocols assure cross-platform portability by using the old algorithm.

  (Contributed by Eric Smith and Mark Dickinson; issue 1580)

New, Improved, and Deprecated Modules

- Added a `collections.Counter` class to support convenient counting of unique items in a sequence or iterable:

```
>>> Counter(['red', 'blue', 'red', 'green', 'blue', 'blue'])
Counter({'blue': 3, 'red': 2, 'green': 1})
```

  (Contributed by Raymond Hettinger; issue 1696199.)

- Added a new module, `tkinter.ttk` for access to the Tk themed widget set. The basic idea of ttk is to separate, to the extent possible, the code implementing a widget's behavior from the code implementing its appearance.

  (Contributed by Guilherme Polo; issue 2983.)

- The `gzip.GzipFile` and `bz2.BZ2File` classes now support the context manager protocol:

```
>>> # Automatically close file after writing
>>> with gzip.GzipFile(filename, "wb") as f:
...     f.write(b"xxx")
```

  (Contributed by Antoine Pitrou.)

- The `decimal` module now supports methods for creating a decimal object from a binary `float`. The conversion is exact but can sometimes be surprising:

```
>>> Decimal.from_float(1.1)
Decimal('1.100000000000000088817841970012523233890533447265625')
```

  The long decimal result shows the actual binary fraction being stored for *1.1*. The fraction has many digits because *1.1* cannot be exactly represented in binary.

  (Contributed by Raymond Hettinger and Mark Dickinson.)

- The `itertools` module grew two new functions. The `itertools.combinations_with_replacement()` function is one of four for generating combinatorics including permutations and Cartesian products. The `itertools.compress()` function mimics its namesake from APL. Also, the existing `itertools.count()` function now has an optional *step* argument and can accept any type of counting sequence including `fractions.Fraction` and `decimal.Decimal`:

```
>>> [p+q for p,q in combinations_with_replacement('LOVE', 2)]
['LL', 'LO', 'LV', 'LE', 'OO', 'OV', 'OE', 'VV', 'VE', 'EE']
```

```
>>> list(compress(data=range(10), selectors=[0,0,1,1,0,1,0,1,0,0]))
[2, 3, 5, 7]

>>> c = count(start=Fraction(1,2), step=Fraction(1,6))
>>> [next(c), next(c), next(c), next(c)]
[Fraction(1, 2), Fraction(2, 3), Fraction(5, 6), Fraction(1, 1)]
```

(Contributed by Raymond Hettinger.)

- `collections.namedtuple()` now supports a keyword argument *rename* which lets invalid fieldnames be automatically converted to positional names in the form _0, _1, etc. This is useful when the field names are being created by an external source such as a CSV header, SQL field list, or user input:

```
>>> query = input()
SELECT region, dept, count(*) FROM main GROUPBY region, dept

>>> cursor.execute(query)
>>> query_fields = [desc[0] for desc in cursor.description]
>>> UserQuery = namedtuple('UserQuery', query_fields, rename=True)
>>> pprint.pprint([UserQuery(*row) for row in cursor])
[UserQuery(region='South', dept='Shipping', _2=185),
 UserQuery(region='North', dept='Accounting', _2=37),
 UserQuery(region='West', dept='Sales', _2=419)]
```

(Contributed by Raymond Hettinger; issue 1818.)

- The `re.sub()`, `re.subn()` and `re.split()` functions now accept a flags parameter.

(Contributed by Gregory Smith.)

- The `logging` module now implements a simple `logging.NullHandler` class for applications that are not using logging but are calling library code that does. Setting-up a null handler will suppress spurious warnings such as "No handlers could be found for logger foo":

```
>>> h = logging.NullHandler()
>>> logging.getLogger("foo").addHandler(h)
```

(Contributed by Vinay Sajip; issue 4384).

- The `runpy` module which supports the `-m` command line switch now supports the execution of packages by looking for and executing a `__main__` submodule when a package name is supplied.

(Contributed by Andi Vajda; issue 4195.)

- The `pdb` module can now access and display source code loaded via `zipimport` (or any other conformant **PEP 302** loader).

(Contributed by Alexander Belopolsky; issue 4201.)

- `functools.partial` objects can now be pickled.

  (Suggested by Antoine Pitrou and Jesse Noller. Implemented by Jack Diederich; issue 5228.)

- Add `pydoc` help topics for symbols so that `help('@')` works as expected in the interactive environment.

  (Contributed by David Laban; issue 4739.)

- The `unittest` module now supports skipping individual tests or classes of tests. And it supports marking a test as a expected failure, a test that is known to be broken, but shouldn't be counted as a failure on a TestResult:

```
class TestGizmo(unittest.TestCase):

    @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
    def test_gizmo_on_windows(self):
        ...

    @unittest.expectedFailure
    def test_gimzo_without_required_library(self):
        ...
```

  Also, tests for exceptions have been builtout to work with context managers using the `with` statement:

```
def test_division_by_zero(self):
    with self.assertRaises(ZeroDivisionError):
        x / 0
```

  In addition, several new assertion methods were added including `assertSetEqual()`, `assertDictEqual()`, `assertDictContainsSubset()`, `assertListEqual()`, `assertTupleEqual()`, `assertSequenceEqual()`, `assertRaisesRegexp()`, `assertIsNone()`, and `assertIsNotNone()`.

  (Contributed by Benjamin Peterson and Antoine Pitrou.)

- The `io` module has three new constants for the `seek()` method `SEEK_SET`, `SEEK_CUR`, and `SEEK_END`.

- The `sys.version_info` tuple is now a named tuple:

```
>>> sys.version_info
```

```
sys.version_info(major=3, minor=1, micro=0, releaselevel='alpha', serial=2)
```

(Contributed by Ross Light; issue 4285.)

- The `nntplib` and `imaplib` modules now support IPv6.

  (Contributed by Derek Morr; issue 1655 and issue 1664.)

- The `pickle` module has been adapted for better interoperability with Python 2.x when used with protocol 2 or lower. The reorganization of the standard library changed the formal reference for many objects. For example, `__builtin__.set` in Python 2 is called `builtins.set` in Python 3. This change confounded efforts to share data between different versions of Python. But now when protocol 2 or lower is selected, the pickler will automatically use the old Python 2 names for both loading and dumping. This remapping is turned-on by default but can be disabled with the *fix_imports* option:

```
>>> s = {1, 2, 3}
>>> pickle.dumps(s, protocol=0)
b'c__builtin__\nset\np0\n((lp1\nL1L\naL2L\naL3L\natp2\nRp3\n.'
>>> pickle.dumps(s, protocol=0, fix_imports=False)
b'cbuiltins\nset\np0\n((lp1\nL1L\naL2L\naL3L\natp2\nRp3\n.'
```

  An unfortunate but unavoidable side-effect of this change is that protocol 2 pickles produced by Python 3.1 won't be readable with Python 3.0. The latest pickle protocol, protocol 3, should be used when migrating data between Python 3.x implementations, as it doesn't attempt to remain compatible with Python 2.x.

  (Contributed by Alexandre Vassalotti and Antoine Pitrou, issue 6137.)

- A new module, `importlib` was added. It provides a complete, portable, pure Python reference implementation of the `import` statement and its counterpart, the `__import__()` function. It represents a substantial step forward in documenting and defining the actions that take place during imports.

  (Contributed by Brett Cannon.)

Optimizations

Major performance enhancements have been added:

- The new I/O library (as defined in **PEP 3116**) was mostly written in Python and quickly proved to be a problematic bottleneck in Python 3.0. In Python 3.1, the I/O library has been entirely rewritten in C and is 2 to 20 times faster depending on the task at hand. The pure Python version is still available for experimentation purposes through the `_pyio` module.

(Contributed by Amaury Forgeot d'Arc and Antoine Pitrou.)

- Added a heuristic so that tuples and dicts containing only untrackable objects are not tracked by the garbage collector. This can reduce the size of collections and therefore the garbage collection overhead on long-running programs, depending on their particular use of datatypes.

  (Contributed by Antoine Pitrou, issue 4688.)

- Enabling a configure option named `--with-computed-gotos` on compilers that support it (notably: gcc, SunPro, icc), the bytecode evaluation loop is compiled with a new dispatch mechanism which gives speedups of up to 20%, depending on the system, the compiler, and the benchmark.

  (Contributed by Antoine Pitrou along with a number of other participants, issue 4753).

- The decoding of UTF-8, UTF-16 and LATIN-1 is now two to four times faster.

  (Contributed by Antoine Pitrou and Amaury Forgeot d'Arc, issue 4868.)

- The **`json`** module now has a C extension to substantially improve its performance. In addition, the API was modified so that json works only with **`str`**, not with **`bytes`**. That change makes the module closely match the JSON specification which is defined in terms of Unicode.

  (Contributed by Bob Ippolito and converted to Py3.1 by Antoine Pitrou and Benjamin Peterson; issue 4136.)

- Unpickling now interns the attribute names of pickled objects. This saves memory and allows pickles to be smaller.

  (Contributed by Jake McGuire and Antoine Pitrou; issue 5084.)

IDLE

- IDLE's format menu now provides an option to strip trailing whitespace from a source file.

  (Contributed by Roger D. Serwy; issue 5150.)

Build and C API Changes

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

- Integers are now stored internally either in base 2**15 or in base 2**30, the base being determined at build time. Previously, they were always stored in base 2**15. Using base 2**30 gives significant performance improvements on 64-bit machines, but benchmark results on 32-bit machines have been mixed. Therefore, the default is to use base 2**30 on 64-bit machines and base 2**15 on 32-bit machines; on Unix, there's a new configure option `--enable-big-digits` that can be used to override this default.

  Apart from the performance improvements this change should be invisible to end users, with one exception: for testing and debugging purposes there's a new `sys.int_info` that provides information about the internal format, giving the number of bits per digit and the size in bytes of the C type used to store each digit:

  ```
  >>> import sys
  >>> sys.int_info
  sys.int_info(bits_per_digit=30, sizeof_digit=4)
  ```

  (Contributed by Mark Dickinson; issue 4258.)

- The `PyLong_AsUnsignedLongLong()` function now handles a negative *pylong* by raising `OverflowError` instead of `TypeError`.

  (Contributed by Mark Dickinson and Lisandro Dalcrin; issue 5175.)

- Deprecated `PyNumber_Int()`. Use `PyNumber_Long()` instead.

  (Contributed by Mark Dickinson; issue 4910.)

- Added a new `PyOS_string_to_double()` function to replace the deprecated functions `PyOS_ascii_strtod()` and `PyOS_ascii_atof()`.

  (Contributed by Mark Dickinson; issue 5914.)

- Added `PyCapsule` as a replacement for the `PyCObject` API. The principal difference is that the new type has a well defined interface for passing typing safety information and a less complicated signature for calling a destructor. The old type had a problematic API and is now deprecated.

  (Contributed by Larry Hastings; issue 5630.)

Porting to Python 3.1

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

- The new floating point string representations can break existing doctests. For example:

```
def e():
    '''Compute the base of natural logarithms.

    >>> e()
    2.71828182845904510

    '''
    return sum(1/math.factorial(x) for x in reversed(range(30)))

doctest.testmod()

**********************************************************************
Failed example:
    e()
Expected:
    2.71828182845904510
Got:
    2.718281828459045
**********************************************************************
```

- The automatic name remapping in the pickle module for protocol 2 or lower can make Python 3.1 pickles unreadable in Python 3.0. One solution is to use protocol 3. Another solution is to set the *fix_imports* option to **False**. See the discussion above for more details.

What's New In Python 3.2

Author: Raymond Hettinger

This article explains the new features in Python 3.2 as compared to 3.1. It focuses on a few highlights and gives a few examples. For full details, see the Misc/NEWS file.

See also: PEP 392 – Python 3.2 Release Schedule

PEP 384: Defining a Stable ABI

In the past, extension modules built for one Python version were often not usable with other Python versions. Particularly on Windows, every feature release of Python required rebuilding all extension modules that one wanted to use. This requirement was the result of the free access to Python interpreter internals that extension modules could use.

With Python 3.2, an alternative approach becomes available: extension modules which restrict themselves to a limited API (by defining Py_LIMITED_API) cannot use many of the internals, but are constrained to a set of API functions that are promised to be stable for several releases. As a consequence, extension modules built for 3.2 in that mode will also work with 3.3, 3.4, and so on. Extension modules that make use of details of memory structures can still be built, but will need to be recompiled for every feature release.

See also:

PEP 384 – Defining a Stable ABI
PEP written by Martin von Löwis.

PEP 389: Argparse Command Line Parsing Module

A new module for command line parsing, `argparse`, was introduced to overcome the limitations of `optparse` which did not provide support for positional arguments (not just options), subcommands, required options and other common patterns of specifying and validating options.

This module has already had widespread success in the community as a third-party module. Being more fully featured than its predecessor, the `argparse` module is now the preferred module for command-line processing. The older module is still being kept available because of the substantial amount of legacy code that depends on it.

Here's an annotated example parser showing features like limiting results to a set of choices, specifying a *metavar* in the help screen, validating that one or more positional arguments is present, and making a required option:

```
import argparse
parser = argparse.ArgumentParser(
            description = 'Manage servers',         # main description fo
            epilog = 'Tested on Solaris and Linux') # displayed after hel
parser.add_argument('action',                       # argument name
            choices = ['deploy', 'start', 'stop'],  # three allowed value
            help = 'action on each target')         # help msg
parser.add_argument('targets',
            metavar = 'HOSTNAME',                   # var name used in he
            nargs = '+',                            # require one or more
            help = 'url for target machines')       # help msg explanatio
parser.add_argument('-u', '--user',                 # -u or --user option
            required = True,                        # make it a required
            help = 'login as user')
```

Example of calling the parser on a command string:

```
>>> cmd  = 'deploy sneezy.example.com sleepy.example.com -u skycaptain'
>>> result = parser.parse_args(cmd.split())
>>> result.action
'deploy'
>>> result.targets
['sneezy.example.com', 'sleepy.example.com']
>>> result.user
'skycaptain'
```

Example of the parser's automatically generated help:

```
>>> parser.parse_args('-h'.split())

usage: manage_cloud.py [-h] -u USER
                       {deploy,start,stop} HOSTNAME [HOSTNAME ...]

Manage servers
```

```
positional arguments:
  {deploy,start,stop}   action on each target
  HOSTNAME              url for target machines

optional arguments:
  -h, --help            show this help message and exit
  -u USER, --user USER  login as user

Tested on Solaris and Linux
```

An especially nice argparse feature is the ability to define subparsers, each with their own argument patterns and help displays:

```
import argparse
parser = argparse.ArgumentParser(prog='HELM')
subparsers = parser.add_subparsers()

parser_l = subparsers.add_parser('launch', help='Launch Control')   # firs
parser_l.add_argument('-m', '--missiles', action='store_true')
parser_l.add_argument('-t', '--torpedos', action='store_true')

parser_m = subparsers.add_parser('move', help='Move Vessel',        # seco
                                 aliases=('steer', 'turn'))         # equi
parser_m.add_argument('-c', '--course', type=int, required=True)
parser_m.add_argument('-s', '--speed', type=int, default=0)

$ ./helm.py --help                         # top level help (launch and mo
$ ./helm.py launch --help                  # help for launch options
$ ./helm.py launch --missiles              # set missiles=True and torpedo
$ ./helm.py steer --course 180 --speed 5   # set movement parameters
```

See also:

PEP 389 – New Command Line Parsing Module
PEP written by Steven Bethard.

Upgrading optparse code for details on the differences from optparse.

PEP 391: Dictionary Based Configuration for Logging

The logging module provided two kinds of configuration, one style with function calls for each option or another style driven by an external file saved in a `ConfigParser` format. Those options did not provide the flexibility to create configurations from JSON or YAML files, nor did they support incremental configuration, which is needed for specifying logger options from a command line.

To support a more flexible style, the module now offers `logging.config.dictConfig()` for specifying logging configuration with plain Python dictionaries. The configuration options include formatters, handlers, filters, and loggers. Here's a working example of a configuration dictionary:

```
{"version": 1,
 "formatters": {"brief": {"format": "%(levelname)-8s: %(name)-15s: %(messa
                "full": {"format": "%(asctime)s %(name)-15s %(levelname)-8
                },
 "handlers": {"console": {
                   "class": "logging.StreamHandler",
                   "formatter": "brief",
                   "level": "INFO",
                   "stream": "ext://sys.stdout"},
              "console_priority": {
                   "class": "logging.StreamHandler",
                   "formatter": "full",
                   "level": "ERROR",
                   "stream": "ext://sys.stderr"}
              },
 "root": {"level": "DEBUG", "handlers": ["console", "console_priority"]}}
```

If that dictionary is stored in a file called `conf.json`, it can be loaded and called with code like this:

```
>>> import json, logging.config
>>> with open('conf.json') as f:
        conf = json.load(f)
>>> logging.config.dictConfig(conf)
>>> logging.info("Transaction completed normally")
INFO    : root           : Transaction completed normally
>>> logging.critical("Abnormal termination")
2011-02-17 11:14:36,694 root            CRITICAL Abnormal termination
```

See also:

PEP 391 – Dictionary Based Configuration for Logging
PEP written by Vinay Sajip.

PEP 3148: The `concurrent.futures` module

Code for creating and managing concurrency is being collected in a new top-level namespace, *concurrent*. Its first member is a *futures* package which provides a uniform high-level interface for managing threads and processes.

The design for `concurrent.futures` was inspired by the *java.util.concurrent* package. In that model, a running call and its result are represented by a `Future` object that abstracts features common to threads, processes, and remote procedure calls. That object supports status checks (running or done), timeouts, cancellations, adding callbacks, and access to results or exceptions.

The primary offering of the new module is a pair of executor classes for launching and managing calls. The goal of the executors is to make it easier to use existing tools for making parallel calls. They save the effort needed to setup a pool of resources, launch the calls, create a results queue, add time-out handling, and limit the total number of threads, processes, or remote procedure calls.

Ideally, each application should share a single executor across multiple components so that process and thread limits can be centrally managed. This solves the design challenge that arises when each component has its own competing strategy for resource management.

Both classes share a common interface with three methods: `submit()` for scheduling a callable and returning a `Future` object; `map()` for scheduling many asynchronous calls at a time, and `shutdown()` for freeing resources. The class is a *context manager* and can be used in a `with` statement to assure that resources are automatically released when currently pending futures are done executing.

A simple of example of `ThreadPoolExecutor` is a launch of four parallel threads for copying files:

```python
import concurrent.futures, shutil
with concurrent.futures.ThreadPoolExecutor(max_workers=4) as e:
    e.submit(shutil.copy, 'src1.txt', 'dest1.txt')
    e.submit(shutil.copy, 'src2.txt', 'dest2.txt')
    e.submit(shutil.copy, 'src3.txt', 'dest3.txt')
    e.submit(shutil.copy, 'src4.txt', 'dest4.txt')
```

See also:

PEP 3148 – Futures – Execute Computations Asynchronously
PEP written by Brian Quinlan.

Code for Threaded Parallel URL reads, an example using threads to fetch multiple web pages in parallel.

Code for computing prime numbers in parallel, an example demonstrating `ProcessPoolExecutor`.

PEP 3147: PYC Repository Directories

Python's scheme for caching bytecode in *.pyc* files did not work well in environments with multiple Python interpreters. If one interpreter encountered a cached file created by another interpreter, it would recompile the source and overwrite the cached file, thus losing the benefits of caching.

The issue of "pyc fights" has become more pronounced as it has become commonplace for Linux distributions to ship with multiple versions of Python. These conflicts also arise with CPython alternatives such as Unladen Swallow.

To solve this problem, Python's import machinery has been extended to use distinct filenames for each interpreter. Instead of Python 3.2 and Python 3.3 and Unladen Swallow each competing for a file called "mymodule.pyc", they will now look for "mymodule.cpython-32.pyc", "mymodule.cpython-33.pyc", and "mymodule.unladen10.pyc". And to prevent all of these new files from cluttering source directories, the *pyc* files are now collected in a "__pycache__" directory stored under the package directory.

Aside from the filenames and target directories, the new scheme has a few aspects that are visible to the programmer:

- Imported modules now have a `__cached__` attribute which stores the name of the actual file that was imported:

```
>>> import collections
>>> collections.__cached__
'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
```

- The tag that is unique to each interpreter is accessible from the `imp` module:

```
>>> import imp
>>> imp.get_tag()
'cpython-32'
```

- Scripts that try to deduce source filename from the imported file now need to be smarter. It is no longer sufficient to simply strip the "c" from a ".pyc" filename. Instead, use the new functions in the `imp` module:

```
>>> imp.source_from_cache('c:/py32/lib/__pycache__/collections.cpytho
'c:/py32/lib/collections.py'
>>> imp.cache_from_source('c:/py32/lib/collections.py')
'c:/py32/lib/__pycache__/collections.cpython-32.pyc'
```

- The `py_compile` and `compileall` modules have been updated to reflect the new naming convention and target directory. The command-line invocation of *compileall* has new options: `-i` for specifying a list of files and directories to compile and `-b` which causes bytecode files to be written to their legacy location rather than *__pycache__*.

- The `importlib.abc` module has been updated with new *abstract base classes* for loading bytecode files. The obsolete ABCs, `PyLoader` and `PyPycLoader`, have been deprecated (instructions on how to stay Python 3.1 compatible are included with the documentation).

See also:

PEP 3147 – PYC Repository Directories
PEP written by Barry Warsaw.

PEP 3149: ABI Version Tagged .so Files

The PYC repository directory allows multiple bytecode cache files to be co-located. This PEP implements a similar mechanism for shared object files by giving them a common directory and distinct names for each version.

The common directory is "pyshared" and the file names are made distinct by identifying the Python implementation (such as CPython, PyPy, Jython, etc.), the major and minor version numbers, and optional build flags (such as "d" for debug, "m" for pymalloc, "u" for wide-unicode). For an arbitrary package "foo", you may see these files when the distribution package is installed:

```
/usr/share/pyshared/foo.cpython-32m.so
/usr/share/pyshared/foo.cpython-33md.so
```

In Python itself, the tags are accessible from functions in the `sysconfig` module:

```
>>> import sysconfig
>>> sysconfig.get_config_var('SOABI')       # find the version tag
'cpython-32mu'
>>> sysconfig.get_config_var('EXT_SUFFIX')  # find the full filename exter
'.cpython-32mu.so'
```

See also:

PEP 3149 – ABI Version Tagged .so Files
PEP written by Barry Warsaw.

PEP 3333: Python Web Server Gateway Interface v1.0.1

This informational PEP clarifies how bytes/text issues are to be handled by the WSGI protocol. The challenge is that string handling in Python 3 is most conveniently handled with the `str` type even though the HTTP protocol is itself bytes oriented.

The PEP differentiates so-called *native strings* that are used for request/response headers and metadata versus *byte strings* which are used for the bodies of requests and responses.

The *native strings* are always of type `str` but are restricted to code points between *U+0000* through *U+00FF* which are translatable to bytes using *Latin-1* encoding. These strings are used for the keys and values in the environment dictionary and for response headers and statuses in the `start_response()` function. They must follow **RFC 2616** with respect to encoding. That is, they must either be *ISO-8859-1* characters or use **RFC 2047** MIME encoding.

For developers porting WSGI applications from Python 2, here are the salient points:

- If the app already used strings for headers in Python 2, no change is needed.
- If instead, the app encoded output headers or decoded input headers, then the headers will need to be re-encoded to Latin-1. For example, an output header encoded in utf-8 was using `h.encode('utf-8')` now needs to convert from bytes to native strings using `h.encode('utf-8').decode('latin-1')`.
- Values yielded by an application or sent using the `write()` method must be byte strings. The `start_response()` function and environ must use native strings. The two cannot be mixed.

For server implementers writing CGI-to-WSGI pathways or other CGI-style protocols, the users must to be able access the environment using native strings even though the underlying platform may have a different convention. To bridge this gap, the `wsgiref` module has a new function, `wsgiref.handlers.read_environ()` for transcoding CGI variables from `os.environ` into native strings and returning a new dictionary.

See also:

PEP 3333 - Python Web Server Gateway Interface v1.0.1
PEP written by Phillip Eby.

Other Language Changes

Some smaller changes made to the core Python language are:

- String formatting for `format()` and `str.format()` gained new capabilities for the format character #. Previously, for integers in binary, octal, or hexadecimal, it caused the output to be prefixed with '0b', '0o', or '0x' respectively. Now it can also handle floats, complex, and Decimal, causing the output to always have a decimal point even when no digits follow it.

```
>>> format(20, '#o')
'0o24'
>>> format(12.34, '#5.0f')
'  12.'
```

  (Suggested by Mark Dickinson and implemented by Eric Smith in issue 7094.)

- There is also a new `str.format_map()` method that extends the capabilities of the existing `str.format()` method by accepting arbitrary *mapping* objects. This new method makes it possible to use string formatting with any of Python's many dictionary-like objects such as `defaultdict`, `Shelf`, `ConfigParser`, or `dbm`. It is also useful with custom `dict` subclasses that normalize keys before look-up or that supply a `__missing__()` method for unknown keys:

```
>>> import shelve
>>> d = shelve.open('tmp.shl')
>>> 'The {project_name} status is {status} as of {date}'.format_map(d
'The testing project status is green as of February 15, 2011'

>>> class LowerCasedDict(dict):
        def __getitem__(self, key):
            return dict.__getitem__(self, key.lower())
>>> lcd = LowerCasedDict(part='widgets', quantity=10)
>>> 'There are {QUANTITY} {Part} in stock'.format_map(lcd)
'There are 10 widgets in stock'

>>> class PlaceholderDict(dict):
        def __missing__(self, key):
            return '<{}>'.format(key)
>>> 'Hello {name}, welcome to {location}'.format_map(PlaceholderDict(
'Hello <name>, welcome to <location>'
```

(Suggested by Raymond Hettinger and implemented by Eric Smith in issue 6081.)

- The interpreter can now be started with a quiet option, `-q`, to prevent the copyright and version information from being displayed in the interactive mode. The option can be introspected using the `sys.flags` attribute:

```
$ python -q
>>> sys.flags
sys.flags(debug=0, division_warning=0, inspect=0, interactive=0,
optimize=0, dont_write_bytecode=0, no_user_site=0, no_site=0,
ignore_environment=0, verbose=0, bytes_warning=0, quiet=1)
```

(Contributed by Marcin Wojdyr in issue 1772833).

- The `hasattr()` function works by calling `getattr()` and detecting whether an exception is raised. This technique allows it to detect methods created dynamically by `__getattr__()` or `__getattribute__()` which would otherwise be absent from the class dictionary. Formerly, *hasattr* would catch any exception, possibly masking genuine errors. Now, *hasattr* has been tightened to only catch `AttributeError` and let other exceptions pass through:

```
>>> class A:
        @property
        def f(self):
            return 1 // 0

>>> a = A()
>>> hasattr(a, 'f')
Traceback (most recent call last):
  ...
ZeroDivisionError: integer division or modulo by zero
```

(Discovered by Yury Selivanov and fixed by Benjamin Peterson; issue 9666.)

- The `str()` of a float or complex number is now the same as its `repr()`. Previously, the `str()` form was shorter but that just caused confusion and is no longer needed now that the shortest possible `repr()` is displayed by default:

```
>>> import math
>>> repr(math.pi)
'3.141592653589793'
>>> str(math.pi)
'3.141592653589793'
```

(Proposed and implemented by Mark Dickinson; issue 9337.)

- `memoryview` objects now have a `release()` method and they also now support the context manager protocol. This allows timely release of any resources that were acquired when requesting a buffer from the original object.

```
>>> with memoryview(b'abcdefgh') as v:
        print(v.tolist())
[97, 98, 99, 100, 101, 102, 103, 104]
```

(Added by Antoine Pitrou; issue 9757.)

- Previously it was illegal to delete a name from the local namespace if it occurs as a free variable in a nested block:

```
def outer(x):
    def inner():
        return x
    inner()
    del x
```

This is now allowed. Remember that the target of an `except` clause is cleared, so this code which used to work with Python 2.6, raised a `SyntaxError` with Python 3.1 and now works again:

```
def f():
    def print_error():
        print(e)
    try:
        something
    except Exception as e:
        print_error()
        # implicit "del e" here
```

(See issue 4617.)

- The internal `structsequence` tool now creates subclasses of tuple. This means that C structures like those returned by `os.stat()`, `time.gmtime()`, and `sys.version_info` now work like a *named tuple* and now work with functions and methods that expect a tuple as an argument. This is a big step forward in making the C structures as flexible as their pure Python counterparts:

```
>>> isinstance(sys.version_info, tuple)
True
>>> 'Version %d.%d.%d %s(%d)' % sys.version_info
'Version 3.2.0 final(0)'
```

(Suggested by Arfrever Frehtes Taifersar Arahesis and implemented by Benjamin Peterson in issue 8413.)

- Warnings are now easier to control using the `PYTHONWARNINGS` environment variable as an alternative to using `-W` at the command line:

```
$ export PYTHONWARNINGS='ignore::RuntimeWarning::,once::UnicodeWarnin
```

(Suggested by Barry Warsaw and implemented by Philip Jenvey in issue 7301.)

- A new warning category, `ResourceWarning`, has been added. It is emitted when potential issues with resource consumption or cleanup are detected. It is silenced by default in normal release builds but can be enabled through the means provided by the `warnings` module, or on the command line.

A `ResourceWarning` is issued at interpreter shutdown if the `gc.garbage` list isn't empty, and if `gc.DEBUG_UNCOLLECTABLE` is set, all uncollectable objects are printed. This is meant to make the programmer aware that their code contains object finalization issues.

A `ResourceWarning` is also issued when a *file object* is destroyed without having been explicitly closed. While the deallocator for such object ensures it closes the underlying operating system resource (usually, a file descriptor), the delay in deallocating the object could produce various issues, especially under Windows. Here is an example of enabling the warning from the command line:

```
$ python -q -Wdefault
>>> f = open("foo", "wb")
>>> del f
__main__:1: ResourceWarning: unclosed file <_io.BufferedWriter name='
```

(Added by Antoine Pitrou and Georg Brandl in issue 10093 and issue 477863.)

- `range` objects now support *index* and *count* methods. This is part of an effort to make more objects fully implement the `collections.Sequence` *abstract base class*. As a result, the language will have a more uniform API. In addition, `range` objects now support slicing and negative indices, even with values larger than `sys.maxsize`. This makes *range* more interoperable with lists:

```
>>> range(0, 100, 2).count(10)
1
>>> range(0, 100, 2).index(10)
5
>>> range(0, 100, 2)[5]
10
>>> range(0, 100, 2)[0:5]
range(0, 10, 2)
```

(Contributed by Daniel Stutzbach in issue 9213, by Alexander Belopolsky in issue 2690, and by Nick Coghlan in issue 10889.)

- The `callable()` builtin function from Py2.x was resurrected. It provides a concise, readable alternative to using an *abstract base class* in an expression like `isinstance(x, collections.Callable)`:

```
>>> callable(max)
True
>>> callable(20)
False
```

(See issue 10518.)

- Python's import mechanism can now load modules installed in directories with non-ASCII characters in the path name. This solved an aggravating problem with home directories for users with non-ASCII characters in their usernames.

  (Required extensive work by Victor Stinner in issue 9425.)

New, Improved, and Deprecated Modules

Python's standard library has undergone significant maintenance efforts and quality improvements.

The biggest news for Python 3.2 is that the `email` package, `mailbox` module, and `nntplib` modules now work correctly with the bytes/text model in Python 3. For the first time, there is correct handling of messages with mixed encodings.

Throughout the standard library, there has been more careful attention to encodings and text versus bytes issues. In particular, interactions with the operating system are now better able to exchange non-ASCII data using the Windows MBCS encoding, locale-aware encodings, or UTF-8.

Another significant win is the addition of substantially better support for *SSL* connections and security certificates.

In addition, more classes now implement a *context manager* to support convenient and reliable resource clean-up using a `with` statement.

email

The usability of the `email` package in Python 3 has been mostly fixed by the extensive efforts of R. David Murray. The problem was that emails are typically read and stored in the form of `bytes` rather than `str` text, and they may contain multiple encodings within a single email. So, the email package had to be extended to parse and generate email messages in bytes format.

- New functions `message_from_bytes()` and `message_from_binary_file()`, and new classes `BytesFeedParser` and `BytesParser` allow binary message data to be parsed into model objects.
- Given bytes input to the model, `get_payload()` will by default decode a message body that has a *Content-Transfer-Encoding* of *8bit* using the charset specified in the MIME headers and return the resulting string.
- Given bytes input to the model, `Generator` will convert message bodies that have a *Content-Transfer-Encoding* of *8bit* to instead have a *7bit Content-Transfer-Encoding*.

  Headers with unencoded non-ASCII bytes are deemed to be **RFC 2047**-encoded using the *unknown-8bit* character set.
- A new class `BytesGenerator` produces bytes as output, preserving any unchanged non-ASCII data that was present in the input used to build the model, including message bodies with a *Content-Transfer-Encoding* of *8bit*.
- The `smtplib SMTP` class now accepts a byte string for the *msg* argument to the `sendmail()` method, and a new method, `send_message()` accepts a `Message` object and can optionally obtain the *from_addr* and *to_addrs* addresses directly from the object.

(Proposed and implemented by R. David Murray, issue 4661 and issue 10321.)

elementtree

The `xml.etree.ElementTree` package and its `xml.etree.cElementTree` counterpart have been updated to version 1.3.

Several new and useful functions and methods have been added:

- `xml.etree.ElementTree.fromstringlist()` which builds an XML document from a sequence of fragments
- `xml.etree.ElementTree.register_namespace()` for registering a global

namespace prefix
- `xml.etree.ElementTree.tostringlist()` for string representation including all sublists
- `xml.etree.ElementTree.Element.extend()` for appending a sequence of zero or more elements
- `xml.etree.ElementTree.Element.iterfind()` searches an element and subelements
- `xml.etree.ElementTree.Element.itertext()` creates a text iterator over an element and its subelements
- `xml.etree.ElementTree.TreeBuilder.end()` closes the current element
- `xml.etree.ElementTree.TreeBuilder.doctype()` handles a doctype declaration

Two methods have been deprecated:

- `xml.etree.ElementTree.getchildren()` use `list(elem)` instead.
- `xml.etree.ElementTree.getiterator()` use `Element.iter` instead.

For details of the update, see Introducing ElementTree on Fredrik Lundh's website.

(Contributed by Florent Xicluna and Fredrik Lundh, issue 6472.)

functools

- The `functools` module includes a new decorator for caching function calls. `functools.lru_cache()` can save repeated queries to an external resource whenever the results are expected to be the same.

  For example, adding a caching decorator to a database query function can save database accesses for popular searches:

  ```
  >>> import functools
  >>> @functools.lru_cache(maxsize=300)
  >>> def get_phone_number(name):
          c = conn.cursor()
          c.execute('SELECT phonenumber FROM phonelist WHERE name=?', (
          return c.fetchone()[0]
  ```

  ```
  >>> for name in user_requests:
          get_phone_number(name)        # cached lookup
  ```

  To help with choosing an effective cache size, the wrapped function is instrumented for tracking cache statistics:

```
>>> get_phone_number.cache_info()
CacheInfo(hits=4805, misses=980, maxsize=300, currsize=300)
```

If the phonelist table gets updated, the outdated contents of the cache can be cleared with:

```
>>> get_phone_number.cache_clear()
```

(Contributed by Raymond Hettinger and incorporating design ideas from Jim Baker, Miki Tebeka, and Nick Coghlan; see recipe 498245, recipe 577479, issue 10586, and issue 10593.)

- The `functools.wraps()` decorator now adds a `__wrapped__` attribute pointing to the original callable function. This allows wrapped functions to be introspected. It also copies `__annotations__` if defined. And now it also gracefully skips over missing attributes such as `__doc__` which might not be defined for the wrapped callable.

  In the above example, the cache can be removed by recovering the original function:

  ```
  >>> get_phone_number = get_phone_number.__wrapped__    # uncached fun
  ```

  (By Nick Coghlan and Terrence Cole; issue 9567, issue 3445, and issue 8814.)

- To help write classes with rich comparison methods, a new decorator `functools.total_ordering()` will use a existing equality and inequality methods to fill in the remaining methods.

  For example, supplying *__eq__* and *__lt__* will enable `total_ordering()` to fill-in *__le__*, *__gt__* and *__ge__*:

  ```
  @total_ordering
  class Student:
      def __eq__(self, other):
          return ((self.lastname.lower(), self.firstname.lower()) ==
                  (other.lastname.lower(), other.firstname.lower()))
      def __lt__(self, other):
          return ((self.lastname.lower(), self.firstname.lower()) <
  ```

```
                    (other.lastname.lower(), other.firstname.lower()))
```

With the *total_ordering* decorator, the remaining comparison methods are filled in automatically.

(Contributed by Raymond Hettinger.)

- To aid in porting programs from Python 2, the `functools.cmp_to_key()` function converts an old-style comparison function to modern *key function*:

  ```
  >>> # locale-aware sort order
  >>> sorted(iterable, key=cmp_to_key(locale.strcoll))
  ```

  For sorting examples and a brief sorting tutorial, see the Sorting HowTo tutorial.

  (Contributed by Raymond Hettinger.)

itertools

- The `itertools` module has a new `accumulate()` function modeled on APL's *scan* operator and Numpy's *accumulate* function:

  ```
  >>> from itertools import accumulate
  >>> list(accumulate([8, 2, 50]))
  [8, 10, 60]
  ```

  ```
  >>> prob_dist = [0.1, 0.4, 0.2, 0.3]
  >>> list(accumulate(prob_dist))      # cumulative probability distrib
  [0.1, 0.5, 0.7, 1.0]
  ```

  For an example using `accumulate()`, see the *examples for the random module*.

  (Contributed by Raymond Hettinger and incorporating design suggestions from Mark Dickinson.)

collections

- The `collections.Counter` class now has two forms of in-place subtraction, the existing *-=* operator for saturating subtraction and the new `subtract()` method for regular subtraction. The former is suitable for multisets which only have positive counts, and the latter is more suitable for use cases that allow negative counts:

```
>>> tally = Counter(dogs=5, cat=3)
>>> tally -= Counter(dogs=2, cats=8)     # saturating subtraction
>>> tally
Counter({'dogs': 3})
```

```
>>> tally = Counter(dogs=5, cats=3)
>>> tally.subtract(dogs=2, cats=8)      # regular subtraction
>>> tally
Counter({'dogs': 3, 'cats': -5})
```

(Contributed by Raymond Hettinger.)

- The `collections.OrderedDict` class has a new method `move_to_end()` which takes an existing key and moves it to either the first or last position in the ordered sequence.

  The default is to move an item to the last position. This is equivalent of renewing an entry with `od[k] = od.pop(k)`.

  A fast move-to-end operation is useful for resequencing entries. For example, an ordered dictionary can be used to track order of access by aging entries from the oldest to the most recently accessed.

```
>>> d = OrderedDict.fromkeys(['a', 'b', 'X', 'd', 'e'])
>>> list(d)
['a', 'b', 'X', 'd', 'e']
>>> d.move_to_end('X')
>>> list(d)
['a', 'b', 'd', 'e', 'X']
```

  (Contributed by Raymond Hettinger.)

- The `collections.deque` class grew two new methods `count()` and `reverse()` that make them more substitutable for `list` objects:

```
>>> d = deque('simsalabim')
>>> d.count('s')
2
>>> d.reverse()
>>> d
deque(['m', 'i', 'b', 'a', 'l', 'a', 's', 'm', 'i', 's'])
```

(Contributed by Raymond Hettinger.)

threading

The `threading` module has a new `Barrier` synchronization class for making multiple threads wait until all of them have reached a common barrier point. Barriers are useful for making sure that a task with multiple preconditions does not run until all of the predecessor tasks are complete.

Barriers can work with an arbitrary number of threads. This is a generalization of a Rendezvous which is defined for only two threads.

Implemented as a two-phase cyclic barrier, `Barrier` objects are suitable for use in loops. The separate *filling* and *draining* phases assure that all threads get released (drained) before any one of them can loop back and re-enter the barrier. The barrier fully resets after each cycle.

Example of using barriers:

```
from threading import Barrier, Thread

def get_votes(site):
    ballots = conduct_election(site)
    all_polls_closed.wait()        # do not count until all polls are clos
    totals = summarize(ballots)
    publish(site, totals)

all_polls_closed = Barrier(len(sites))
for site in sites:
    Thread(target=get_votes, args=(site,)).start()
```

In this example, the barrier enforces a rule that votes cannot be counted at any polling site until all polls are closed. Notice how a solution with a barrier is similar to one with `threading.Thread.join()`, but the threads stay alive and continue to do work (summarizing ballots) after the barrier point is crossed.

If any of the predecessor tasks can hang or be delayed, a barrier can be created with an optional *timeout* parameter. Then if the timeout period elapses before all the predecessor tasks reach the barrier point, all waiting threads are released and a BrokenBarrierError exception is raised:

```
def get_votes(site):
    ballots = conduct_election(site)
    try:
        all_polls_closed.wait(timeout = midnight - time.now())
    except BrokenBarrierError:
        lockbox = seal_ballots(ballots)
        queue.put(lockbox)
    else:
        totals = summarize(ballots)
        publish(site, totals)
```

In this example, the barrier enforces a more robust rule. If some election sites do not finish before midnight, the barrier times-out and the ballots are sealed and deposited in a queue for later handling.

See Barrier Synchronization Patterns for more examples of how barriers can be used in parallel computing. Also, there is a simple but thorough explanation of barriers in The Little Book of Semaphores, *section 3.6*.

(Contributed by Kristján Valur Jónsson with an API review by Jeffrey Yasskin in issue 8777.)

datetime and time

- The datetime module has a new type timezone that implements the tzinfo interface by returning a fixed UTC offset and timezone name. This makes it easier to create timezone-aware datetime objects:

  ```
  >>> from datetime import datetime, timezone

  >>> datetime.now(timezone.utc)
  datetime.datetime(2010, 12, 8, 21, 4, 2, 923754, tzinfo=datetime.time

  >>> datetime.strptime("01/01/2000 12:00 +0000", "%m/%d/%Y %H:%M %z")
  datetime.datetime(2000, 1, 1, 12, 0, tzinfo=datetime.timezone.utc)
  ```

- Also, timedelta objects can now be multiplied by float and divided by float and int objects. And timedelta objects can now divide one another.

- The `datetime.date.strftime()` method is no longer restricted to years after 1900. The new supported year range is from 1000 to 9999 inclusive.

- Whenever a two-digit year is used in a time tuple, the interpretation has been governed by `time.accept2dyear`. The default is *True* which means that for a two-digit year, the century is guessed according to the POSIX rules governing the `%y` strptime format.

  Starting with Py3.2, use of the century guessing heuristic will emit a `DeprecationWarning`. Instead, it is recommended that `time.accept2dyear` be set to *False* so that large date ranges can be used without guesswork:

  ```
  >>> import time, warnings
  >>> warnings.resetwarnings()      # remove the default warning filter

  >>> time.accept2dyear = True      # guess whether 11 means 11 or 2011
  >>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
  Warning (from warnings module):
    ...
  DeprecationWarning: Century info guessed for a 2-digit year.
  'Fri Jan  1 12:34:56 2011'

  >>> time.accept2dyear = False     # use the full range of allowable d
  >>> time.asctime((11, 1, 1, 12, 34, 56, 4, 1, 0))
  'Fri Jan  1 12:34:56 11'
  ```

  Several functions now have significantly expanded date ranges. When `time.accept2dyear` is false, the `time.asctime()` function will accept any year that fits in a C int, while the `time.mktime()` and `time.strftime()` functions will accept the full range supported by the corresponding operating system functions.

(Contributed by Alexander Belopolsky and Victor Stinner in issue 1289118, issue 5094, issue 6641, issue 2706, issue 1777412, issue 8013, and issue 10827.)

math

The `math` module has been updated with six new functions inspired by the C99 standard.

The `isfinite()` function provides a reliable and fast way to detect special values. It returns *True* for regular numbers and *False* for *Nan* or *Infinity*:

```
>>> [isfinite(x) for x in (123, 4.56, float('Nan'), float('Inf'))]
[True, True, False, False]
```

The `expm1()` function computes `e**x-1` for small values of *x* without incurring the loss of precision that usually accompanies the subtraction of nearly equal quantities:

```
>>> expm1(0.013671875)   # more accurate way to compute e**x-1 for a smal
0.013765762467652909
```

The `erf()` function computes a probability integral or Gaussian error function. The complementary error function, `erfc()`, is `1 - erf(x)`:

```
>>> erf(1.0/sqrt(2.0))   # portion of normal distribution within 1 standa
0.682689492137086
>>> erfc(1.0/sqrt(2.0))  # portion of normal distribution outside 1 standa
0.31731050786291404
>>> erf(1.0/sqrt(2.0)) + erfc(1.0/sqrt(2.0))
1.0
```

The `gamma()` function is a continuous extension of the factorial function. See http://en.wikipedia.org/wiki/Gamma_function for details. Because the function is related to factorials, it grows large even for small values of *x*, so there is also a `lgamma()` function for computing the natural logarithm of the gamma function:

```
>>> gamma(7.0)           # six factorial
720.0
>>> lgamma(801.0)        # log(800 factorial)
4551.950730698041
```

(Contributed by Mark Dickinson.)

abc

The `abc` module now supports `abstractclassmethod()` and `abstractstaticmethod()`.

These tools make it possible to define an *abstract base class* that requires a particular `classmethod()` or `staticmethod()` to be implemented:

```
class Temperature(metaclass=abc.ABCMeta):
    @abc.abstractclassmethod
    def from_fahrenheit(cls, t):
        ...
    @abc.abstractclassmethod
    def from_celsius(cls, t):
        ...
```

(Patch submitted by Daniel Urban; issue 5867.)

io

The `io.BytesIO` has a new method, `getbuffer()`, which provides functionality similar to `memoryview()`. It creates an editable view of the data without making a copy. The buffer’s random access and support for slice notation are well-suited to in-place editing:

```
>>> REC_LEN, LOC_START, LOC_LEN = 34, 7, 11

>>> def change_location(buffer, record_number, location):
        start = record_number * REC_LEN + LOC_START
        buffer[start: start+LOC_LEN] = location

>>> import io

>>> byte_stream = io.BytesIO(
    b'G3805  storeroom  Main chassis    '
    b'X7899  shipping   Reserve cog     '
    b'L6988  receiving  Primary sprocket'
)
>>> buffer = byte_stream.getbuffer()
>>> change_location(buffer, 1, b'warehouse  ')
>>> change_location(buffer, 0, b'showroom   ')
>>> print(byte_stream.getvalue())
b'G3805  showroom   Main chassis    '
b'X7899  warehouse  Reserve cog     '
b'L6988  receiving  Primary sprocket'
```

(Contributed by Antoine Pitrou in issue 5506.)

reprlib

When writing a `__repr__()` method for a custom container, it is easy to forget to handle the case where a member refers back to the container itself. Python's builtin objects such as `list` and `set` handle self-reference by displaying "..." in the recursive part of the representation string.

To help write such `__repr__()` methods, the `reprlib` module has a new decorator, `recursive_repr()`, for detecting recursive calls to `__repr__()` and substituting a placeholder string instead:

```
>>> class MyList(list):
        @recursive_repr()
        def __repr__(self):
            return '<' + '|'.join(map(repr, self)) + '>'

>>> m = MyList('abc')
>>> m.append(m)
>>> m.append('x')
>>> print(m)
<'a'|'b'|'c'|...|'x'>
```

(Contributed by Raymond Hettinger in issue 9826 and issue 9840.)

logging

In addition to dictionary-based configuration described above, the `logging` package has many other improvements.

The logging documentation has been augmented by a *basic tutorial*, an *advanced tutorial*, and a *cookbook* of logging recipes. These documents are the fastest way to learn about logging.

The `logging.basicConfig()` set-up function gained a *style* argument to support three different types of string formatting. It defaults to "%" for traditional %-formatting, can be set to "{" for the new `str.format()` style, or can be set to "$" for the shell-style formatting provided by `string.Template`. The following three configurations are equivalent:

```
>>> from logging import basicConfig
>>> basicConfig(style='%', format="%(name)s -> %(levelname)s: %(message)s'
>>> basicConfig(style='{', format="{name} -> {levelname} {message}")
>>> basicConfig(style='$', format="$name -> $levelname: $message")
```

If no configuration is set-up before a logging event occurs, there is now a default configuration using a `StreamHandler` directed to `sys.stderr` for events of `WARNING` level or higher. Formerly, an event occurring before a configuration was set-up would either raise an exception or silently drop the event depending on the value of `logging.raiseExceptions`. The new default handler is stored in `logging.lastResort`.

The use of filters has been simplified. Instead of creating a `Filter` object, the predicate can be any Python callable that returns *True* or *False*.

There were a number of other improvements that add flexibility and simplify configuration. See the module documentation for a full listing of changes in Python 3.2.

csv

The `csv` module now supports a new dialect, `unix_dialect`, which applies quoting for all fields and a traditional Unix style with `'\n'` as the line terminator. The registered dialect name is `unix`.

The `csv.DictWriter` has a new method, `writeheader()` for writing-out an initial row to document the field names:

```
>>> import csv, sys
>>> w = csv.DictWriter(sys.stdout, ['name', 'dept'], dialect='unix')
>>> w.writeheader()
"name","dept"
>>> w.writerows([
        {'name': 'tom', 'dept': 'accounting'},
        {'name': 'susan', 'dept': 'Salesl'}])
"tom","accounting"
"susan","sales"
```

(New dialect suggested by Jay Talbot in issue 5975, and the new method suggested by Ed Abraham in issue 1537721.)

contextlib

There is a new and slightly mind-blowing tool `ContextDecorator` that is helpful for creating a *context manager* that does double duty as a function decorator.

As a convenience, this new functionality is used by `contextmanager()` so that no extra effort is needed to support both roles.

The basic idea is that both context managers and function decorators can be used for pre-action and post-action wrappers. Context managers wrap a group of statements using a `with` statement, and function decorators wrap a group of statements enclosed in a function. So, occasionally there is a need to write a pre-action or post-action wrapper that can be used in either role.

For example, it is sometimes useful to wrap functions or groups of statements with a logger that can track the time of entry and time of exit. Rather than writing both a function decorator and a context manager for the task, the `contextmanager()` provides both capabilities in a single definition:

```
from contextlib import contextmanager
import logging

logging.basicConfig(level=logging.INFO)

@contextmanager
def track_entry_and_exit(name):
    logging.info('Entering: {}'.format(name))
    yield
    logging.info('Exiting: {}'.format(name))
```

Formerly, this would have only been usable as a context manager:

```
with track_entry_and_exit('widget loader'):
    print('Some time consuming activity goes here')
    load_widget()
```

Now, it can be used as a decorator as well:

```
@track_entry_and_exit('widget loader')
def activity():
    print('Some time consuming activity goes here')
    load_widget()
```

Trying to fulfill two roles at once places some limitations on the technique. Context managers normally have the flexibility to return an argument usable by a `with` statement, but there is no parallel for function decorators.

In the above example, there is not a clean way for the *track_entry_and_exit* context manager to return a logging instance for use in the body of enclosed statements.

(Contributed by Michael Foord in issue 9110.)

decimal and fractions

Mark Dickinson crafted an elegant and efficient scheme for assuring that different numeric datatypes will have the same hash value whenever their actual values are equal (issue 8188):

```
assert hash(Fraction(3, 2)) == hash(1.5) == \
       hash(Decimal("1.5")) == hash(complex(1.5, 0))
```

Some of the hashing details are exposed through a new attribute, `sys.hash_info`, which describes the bit width of the hash value, the prime modulus, the hash values for *infinity* and *nan*, and the multiplier used for the imaginary part of a number:

```
>>> sys.hash_info
sys.hash_info(width=64, modulus=2305843009213693951, inf=314159, nan=0, in
```

An early decision to limit the inter-operability of various numeric types has been relaxed. It is still unsupported (and ill-advised) to have implicit mixing in arithmetic expressions such as `Decimal('1.1') + float('1.1')` because the latter loses information in the process of constructing the binary float. However, since existing floating point value can be converted losslessly to either a decimal or rational representation, it makes sense to add them to the constructor and to support mixed-type comparisons.

- The `decimal.Decimal` constructor now accepts `float` objects directly so there in no longer a need to use the `from_float()` method (issue 8257).
- Mixed type comparisons are now fully supported so that `Decimal` objects can be directly compared with `float` and `fractions.Fraction` (issue 2531 and issue 8188).

Similar changes were made to `fractions.Fraction` so that the `from_float()` and `from_decimal()` methods are no longer needed (issue 8294):

```
>>> Decimal(1.1)
Decimal('1.100000000000000088817841970012523233890533447265625')
```

```
>>> Fraction(1.1)
Fraction(2476979795053773, 2251799813685248)
```

Another useful change for the decimal module is that the Context.clamp attribute is now public. This is useful in creating contexts that correspond to the decimal interchange formats specified in IEEE 754 (see issue 8540).

(Contributed by Mark Dickinson and Raymond Hettinger.)

ftp

The ftplib.FTP class now supports the context manager protocol to unconditionally consume socket.error exceptions and to close the FTP connection when done:

```
>>> from ftplib import FTP
>>> with FTP("ftp1.at.proftpd.org") as ftp:
        ftp.login()
        ftp.dir()

'230 Anonymous login ok, restrictions apply.'
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 .
dr-xr-xr-x   9 ftp      ftp           154 May  6 10:43 ..
dr-xr-xr-x   5 ftp      ftp          4096 May  6 10:43 CentOS
dr-xr-xr-x   3 ftp      ftp            18 Jul 10  2008 Fedora
```

Other file-like objects such as mmap.mmap and fileinput.input() also grew auto-closing context managers:

```
with fileinput.input(files=('log1.txt', 'log2.txt')) as f:
    for line in f:
        process(line)
```

(Contributed by Tarek Ziadé and Giampaolo Rodolà in issue 4972, and by Georg Brandl in issue 8046 and issue 1286.)

The FTP_TLS class now accepts a *context* parameter, which is a ssl.SSLContext object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; issue 8806.)

popen

The `os.popen()` and `subprocess.Popen()` functions now support `with` statements for auto-closing of the file descriptors.

(Contributed by Antoine Pitrou and Brian Curtin in issue 7461 and issue 10554.)

select

The `select` module now exposes a new, constant attribute, `PIPE_BUF`, which gives the minimum number of bytes which are guaranteed not to block when `select.select()` says a pipe is ready for writing.

```
>>> import select
>>> select.PIPE_BUF
512
```

(Available on Unix systems. Patch by Sébastien Sablé in issue 9862)

gzip and zipfile

`gzip.GzipFile` now implements the `io.BufferedIOBase` *abstract base class* (except for `truncate()`). It also has a `peek()` method and supports unseekable as well as zero-padded file objects.

The `gzip` module also gains the `compress()` and `decompress()` functions for easier in-memory compression and decompression. Keep in mind that text needs to be encoded as `bytes` before compressing and decompressing:

```
>>> s = 'Three shall be the number thou shalt count, '
>>> s += 'and the number of the counting shall be three'
>>> b = s.encode()                        # convert to utf-8
>>> len(b)
89
>>> c = gzip.compress(b)
>>> len(c)
77
>>> gzip.decompress(c).decode()[:42]      # decompress and convert to text
'Three shall be the number thou shalt count,'
```

(Contributed by Anand B. Pillai in issue 3488; and by Antoine Pitrou, Nir Aides and Brian Curtin in issue 9962, issue 1675951, issue 7471 and issue 2846.)

Also, the `zipfile.ZipExtFile` class was reworked internally to represent files stored inside an archive. The new implementation is significantly faster and can be wrapped in a `io.BufferedReader` object for more speedups. It also solves an issue where interleaved calls to *read* and *readline* gave the wrong results.

(Patch submitted by Nir Aides in issue 7610.)

tarfile

The `TarFile` class can now be used as a context manager. In addition, its `add()` method has a new option, *filter*, that controls which files are added to the archive and allows the file metadata to be edited.

The new *filter* option replaces the older, less flexible *exclude* parameter which is now deprecated. If specified, the optional *filter* parameter needs to be a *keyword argument*. The user-supplied filter function accepts a `TarInfo` object and returns an updated `TarInfo` object, or if it wants the file to be excluded, the function can return *None*:

```
>>> import tarfile, glob

>>> def myfilter(tarinfo):
       if tarinfo.isfile():             # only save real files
            tarinfo.uname = 'monty'     # redact the user name
            return tarinfo

>>> with tarfile.open(name='myarchive.tar.gz', mode='w:gz') as tf:
        for filename in glob.glob('*.txt'):
            tf.add(filename, filter=myfilter)
        tf.list()
-rw-r--r-- monty/501        902 2011-01-26 17:59:11 annotations.txt
-rw-r--r-- monty/501        123 2011-01-26 17:59:11 general_questions.txt
-rw-r--r-- monty/501       3514 2011-01-26 17:59:11 prion.txt
-rw-r--r-- monty/501        124 2011-01-26 17:59:11 py_todo.txt
-rw-r--r-- monty/501       1399 2011-01-26 17:59:11 semaphore_notes.txt
```

(Proposed by Tarek Ziadé and implemented by Lars Gustäbel in issue 6856.)

hashlib

The hashlib module has two new constant attributes listing the hashing algorithms guaranteed to be present in all implementations and those available on the current implementation:

```
>>> import hashlib

>>> hashlib.algorithms_guaranteed
{'sha1', 'sha224', 'sha384', 'sha256', 'sha512', 'md5'}

>>> hashlib.algorithms_available
{'md2', 'SHA256', 'SHA512', 'dsaWithSHA', 'mdc2', 'SHA224', 'MD4', 'sha256
'sha512', 'ripemd160', 'SHA1', 'MDC2', 'SHA', 'SHA384', 'MD2',
'ecdsa-with-SHA1','md4', 'md5', 'sha1', 'DSA-SHA', 'sha224',
'dsaEncryption', 'DSA', 'RIPEMD160', 'sha', 'MD5', 'sha384'}
```

(Suggested by Carl Chenet in issue 7418.)

ast

The ast module has a wonderful a general-purpose tool for safely evaluating expression strings using the Python literal syntax. The ast.literal_eval() function serves as a secure alternative to the builtin eval() function which is easily abused. Python 3.2 adds bytes and set literals to the list of supported types: strings, bytes, numbers, tuples, lists, dicts, sets, booleans, and None.

```
>>> from ast import literal_eval

>>> request = "{'req': 3, 'func': 'pow', 'args': (2, 0.5)}"
>>> literal_eval(request)
{'args': (2, 0.5), 'req': 3, 'func': 'pow'}

>>> request = "os.system('do something harmful')"
>>> literal_eval(request)
Traceback (most recent call last):
  ...
ValueError: malformed node or string: <_ast.Call object at 0x101739a10>
```

(Implemented by Benjamin Peterson and Georg Brandl.)

os

Different operating systems use various encodings for filenames and environment variables. The `os` module provides two new functions, `fsencode()` and `fsdecode()`, for encoding and decoding filenames:

```
>>> filename = 'Sehenswürdigkeiten'
>>> os.fsencode(filename)
b'Sehensw\xc3\xbcrdigkeiten'
```

Some operating systems allow direct access to encoded bytes in the environment. If so, the `os.supports_bytes_environ` constant will be true.

For direct access to encoded environment variables (if available), use the new `os.getenvb()` function or use `os.environb` which is a bytes version of `os.environ`.

(Contributed by Victor Stinner.)

shutil

The `shutil.copytree()` function has two new options:

- *ignore_dangling_symlinks*: when `symlinks=False` so that the function copies a file pointed to by a symlink, not the symlink itself. This option will silence the error raised if the file doesn't exist.
- *copy_function*: is a callable that will be used to copy files. `shutil.copy2()` is used by default.

(Contributed by Tarek Ziadé.)

In addition, the `shutil` module now supports *archiving operations* for zipfiles, uncompressed tarfiles, gzipped tarfiles, and bzipped tarfiles. And there are functions for registering additional archiving file formats (such as xz compressed tarfiles or custom formats).

The principal functions are `make_archive()` and `unpack_archive()`. By default, both operate on the current directory (which can be set by `os.chdir()`) and on any sub-directories. The archive filename needs to be specified with a full pathname. The archiving step is non-destructive (the original files are left unchanged).

```
>>> import shutil, pprint

>>> os.chdir('mydata')                               # change to the sourc
>>> f = shutil.make_archive('/var/backup/mydata',
                            'zip')                   # archive the current
>>> f                                                # show the name of ar
'/var/backup/mydata.zip'
>>> os.chdir('tmp')                                  # change to an unpack
>>> shutil.unpack_archive('/var/backup/mydata.zip')  # recover the data

>>> pprint.pprint(shutil.get_archive_formats())      # display known forma
[('bztar', "bzip2'ed tar-file"),
 ('gztar', "gzip'ed tar-file"),
 ('tar', 'uncompressed tar file'),
 ('zip', 'ZIP file')]

>>> shutil.register_archive_format(                  # register a new arch
        name = 'xz',
        function = xz.compress,                      # callable archiving
        extra_args = [('level', 8)],                 # arguments to the fu
        description = 'xz compression'
)
```

(Contributed by Tarek Ziadé.)

sqlite3

The `sqlite3` module was updated to pysqlite version 2.6.0. It has two new capabilities.

- The `sqlite3.Connection.in_transit` attribute is true if there is an active transaction for uncommitted changes.
- The `sqlite3.Connection.enable_load_extension()` and `sqlite3.Connection.load_extension()` methods allows you to load SQLite extensions from ".so" files. One well-known extension is the fulltext-search extension distributed with SQLite.

(Contributed by R. David Murray and Shashwat Anand; issue 8845.)

html

A new `html` module was introduced with only a single function, `escape()`, which is used for escaping reserved characters from HTML markup:

```
>>> import html
>>> html.escape('x > 2 && x < 7')
'x &gt; 2 &amp;&amp; x &lt; 7'
```

socket

The `socket` module has two new improvements.

- Socket objects now have a `detach()` method which puts the socket into closed state without actually closing the underlying file descriptor. The latter can then be reused for other purposes. (Added by Antoine Pitrou; issue 8524.)
- `socket.create_connection()` now supports the context manager protocol to unconditionally consume `socket.error` exceptions and to close the socket when done. (Contributed by Giampaolo Rodolà; issue 9794.)

ssl

The `ssl` module added a number of features to satisfy common requirements for secure (encrypted, authenticated) internet connections:

- A new class, `SSLContext`, serves as a container for persistent SSL data, such as protocol settings, certificates, private keys, and various other options. It includes a `wrap_socket()` for creating an SSL socket from an SSL context.
- A new function, `ssl.match_hostname()`, supports server identity verification for higher-level protocols by implementing the rules of HTTPS (from **RFC 2818**) which are also suitable for other protocols.
- The `ssl.wrap_socket()` constructor function now takes a *ciphers* argument. The *ciphers* string lists the allowed encryption algorithms using the format described in the OpenSSL documentation.
- When linked against recent versions of OpenSSL, the `ssl` module now supports the Server Name Indication extension to the TLS protocol, allowing multiple "virtual hosts" using different certificates on a single IP port. This extension is only supported in client mode, and is activated by passing the *server_hostname* argument to `ssl.SSLContext.wrap_socket()`.
- Various options have been added to the `ssl` module, such as `OP_NO_SSLv2` which disables the insecure and obsolete SSLv2 protocol.
- The extension now loads all the OpenSSL ciphers and digest algorithms. If some SSL certificates cannot be verified, they are reported as an "unknown algorithm"

error.
- The version of OpenSSL being used is now accessible using the module attributes `ssl.OPENSSL_VERSION` (a string), `ssl.OPENSSL_VERSION_INFO` (a 5-tuple), and `ssl.OPENSSL_VERSION_NUMBER` (an integer).

(Contributed by Antoine Pitrou in issue 8850, issue 1589, issue 8322, issue 5639, issue 4870, issue 8484, and issue 8321.)

nntp

The `nntplib` module has a revamped implementation with better bytes and text semantics as well as more practical APIs. These improvements break compatibility with the nntplib version in Python 3.1, which was partly dysfunctional in itself.

Support for secure connections through both implicit (using `nntplib.NNTP_SSL`) and explicit (using `nntplib.NNTP.starttls()`) TLS has also been added.

(Contributed by Antoine Pitrou in issue 9360 and Andrew Vant in issue 1926.)

certificates

`http.client.HTTPSConnection`, `urllib.request.HTTPSHandler` and `urllib.request.urlopen()` now take optional arguments to allow for server certificate checking against a set of Certificate Authorities, as recommended in public uses of HTTPS.

(Added by Antoine Pitrou, issue 9003.)

imaplib

Support for explicit TLS on standard IMAP4 connections has been added through the new `imaplib.IMAP4.starttls` method.

(Contributed by Lorenzo M. Catucci and Antoine Pitrou, issue 4471.)

http.client

There were a number of small API improvements in the `http.client` module. The old-style HTTP 0.9 simple responses are no longer supported and the *strict* parameter is deprecated in all classes.

The `HTTPConnection` and `HTTPSConnection` classes now have a *source_address* parameter for a (host, port) tuple indicating where the HTTP connection is made from.

Support for certificate checking and HTTPS virtual hosts were added to `HTTPSConnection`.

The `request()` method on connection objects allowed an optional *body* argument so that a *file object* could be used to supply the content of the request. Conveniently, the *body* argument now also accepts an *iterable* object so long as it includes an explicit `Content-Length` header. This extended interface is much more flexible than before.

To establish an HTTPS connection through a proxy server, there is a new `set_tunnel()` method that sets the host and port for HTTP Connect tunneling.

To match the behavior of `http.server`, the HTTP client library now also encodes headers with ISO-8859-1 (Latin-1) encoding. It was already doing that for incoming headers, so now the behavior is consistent for both incoming and outgoing traffic. (See work by Armin Ronacher in issue 10980.)

unittest

The unittest module has a number of improvements supporting test discovery for packages, easier experimentation at the interactive prompt, new testcase methods, improved diagnostic messages for test failures, and better method names.

- The command-line call `python -m unittest` can now accept file paths instead of module names for running specific tests (issue 10620). The new test discovery can find tests within packages, locating any test importable from the top-level directory. The top-level directory can be specified with the *-t* option, a pattern for matching files with `-p`, and a directory to start discovery with `-s`:

  ```
  $ python -m unittest discover -s my_proj_dir -p _test.py
  ```

  (Contributed by Michael Foord.)

- Experimentation at the interactive prompt is now easier because the unittest.case.TestCase class can now be instantiated without arguments:

```
>>> TestCase().assertEqual(pow(2, 3), 8)
```

(Contributed by Michael Foord.)

- The unittest module has two new methods, assertWarns() and assertWarnsRegex() to verify that a given warning type is triggered by the code under test:

  ```
  with self.assertWarns(DeprecationWarning):
      legacy_function('XYZ')
  ```

  (Contributed by Antoine Pitrou, issue 9754.)

  Another new method, assertCountEqual() is used to compare two iterables to determine if their element counts are equal (whether the same elements are present with the same number of occurrences regardless of order):

  ```
  def test_anagram(self):
      self.assertCountEqual('algorithm', 'logarithm')
  ```

  (Contributed by Raymond Hettinger.)

- A principal feature of the unittest module is an effort to produce meaningful diagnostics when a test fails. When possible, the failure is recorded along with a diff of the output. This is especially helpful for analyzing log files of failed test runs. However, since diffs can sometime be voluminous, there is a new maxDiff attribute that sets maximum length of diffs displayed.

- In addition, the method names in the module have undergone a number of clean-ups.

  For example, assertRegex() is the new name for assertRegexpMatches() which was misnamed because the test uses re.search(), not re.match(). Other methods using regular expressions are now named using short form "Regex" in preference to "Regexp" – this matches the names used in other unittest implementations, matches Python's old name for the re module, and it has unambiguous camel-casing.

(Contributed by Raymond Hettinger and implemented by Ezio Melotti.)

- To improve consistency, some long-standing method aliases are being deprecated in favor of the preferred names:

Old Name	Preferred Name
`assert_()`	`assertTrue()`
`assertEquals()`	`assertEqual()`
`assertNotEquals()`	`assertNotEqual()`
`assertAlmostEquals()`	`assertAlmostEqual()`
`assertNotAlmostEquals()`	`assertNotAlmostEqual()`

  Likewise, the `TestCase.fail*` methods deprecated in Python 3.1 are expected to be removed in Python 3.3. Also see the *Deprecated aliases* section in the `unittest` documentation.

  (Contributed by Ezio Melotti; issue 9424.)

- The `assertDictContainsSubset()` method was deprecated because it was misimplemented with the arguments in the wrong order. This created hard-to-debug optical illusions where tests like `TestCase().assertDictContainsSubset({'a':1, 'b':2}, {'a':1})` would fail.

  (Contributed by Raymond Hettinger.)

random

The integer methods in the `random` module now do a better job of producing uniform distributions. Previously, they computed selections with `int(n*random())` which had a slight bias whenever *n* was not a power of two. Now, multiple selections are made from a range up to the next power of two and a selection is kept only when it falls within the range `0 <= x < n`. The functions and methods affected are `randrange()`, `randint()`, `choice()`, `shuffle()` and `sample()`.

(Contributed by Raymond Hettinger; issue 9025.)

poplib

POP3_SSL class now accepts a *context* parameter, which is a ssl.SSLContext object allowing bundling SSL configuration options, certificates and private keys into a single (potentially long-lived) structure.

(Contributed by Giampaolo Rodolà; issue 8807.)

asyncore

asyncore.dispatcher now provides a handle_accepted() method returning a *(sock, addr)* pair which is called when a connection has actually been established with a new remote endpoint. This is supposed to be used as a replacement for old handle_accept() and avoids the user to call accept() directly.

(Contributed by Giampaolo Rodolà; issue 6706.)

tempfile

The tempfile module has a new context manager, TemporaryDirectory which provides easy deterministic cleanup of temporary directories:

```
with tempfile.TemporaryDirectory() as tmpdirname:
    print('created temporary dir:', tmpdirname)
```

(Contributed by Neil Schemenauer and Nick Coghlan; issue 5178.)

inspect

- The inspect module has a new function getgeneratorstate() to easily identify the current state of a generator-iterator:

```
>>> from inspect import getgeneratorstate
>>> def gen():
        yield 'demo'
>>> g = gen()
>>> getgeneratorstate(g)
'GEN_CREATED'
>>> next(g)
'demo'
>>> getgeneratorstate(g)
```

```
'GEN_SUSPENDED'
>>> next(g, None)
>>> getgeneratorstate(g)
'GEN_CLOSED'
```

(Contributed by Rodolpho Eckhardt and Nick Coghlan, issue 10220.)

- To support lookups without the possibility of activating a dynamic attribute, the `inspect` module has a new function, `getattr_static()`. Unlike `hasattr()`, this is a true read-only search, guaranteed not to change state while it is searching:

```
>>> class A:
        @property
        def f(self):
            print('Running')
            return 10

>>> a = A()
>>> getattr(a, 'f')
Running
10
>>> inspect.getattr_static(a, 'f')
<property object at 0x1022bd788>
```

(Contributed by Michael Foord.)

pydoc

The `pydoc` module now provides a much-improved Web server interface, as well as a new command-line option `-b` to automatically open a browser window to display that server:

```
$ pydoc3.2 -b
```

(Contributed by Ron Adam; issue 2001.)

dis

The `dis` module gained two new functions for inspecting code, `code_info()` and `show_code()`. Both provide detailed code object information for the supplied function, method, source code string or code object. The former returns a string and the latter prints it:

```
>>> import dis, random
>>> dis.show_code(random.choice)
Name:              choice
Filename:          /Library/Frameworks/Python.framework/Versions/3.2/lib/p
Argument count:    2
Kw-only arguments: 0
Number of locals:  3
Stack size:        11
Flags:             OPTIMIZED, NEWLOCALS, NOFREE
Constants:
   0: 'Choose a random element from a non-empty sequence.'
   1: 'Cannot choose from an empty sequence'
Names:
   0: _randbelow
   1: len
   2: ValueError
   3: IndexError
Variable names:
   0: self
   1: seq
   2: i
```

In addition, the `dis()` function now accepts string arguments so that the common idiom `dis(compile(s, '', 'eval'))` can be shortened to `dis(s)`:

```
>>> dis('3*x+1 if x%2==1 else x//2')
  1           0 LOAD_NAME                0 (x)
              3 LOAD_CONST               0 (2)
              6 BINARY_MODULO
              7 LOAD_CONST               1 (1)
             10 COMPARE_OP               2 (==)
             13 POP_JUMP_IF_FALSE       28
             16 LOAD_CONST               2 (3)
             19 LOAD_NAME                0 (x)
             22 BINARY_MULTIPLY
             23 LOAD_CONST               1 (1)
             26 BINARY_ADD
             27 RETURN_VALUE
        >>   28 LOAD_NAME                0 (x)
             31 LOAD_CONST               0 (2)
             34 BINARY_FLOOR_DIVIDE
             35 RETURN_VALUE
```

Taken together, these improvements make it easier to explore how CPython is implemented and to see for yourself what the language syntax does under-the-hood.

(Contributed by Nick Coghlan in issue 9147.)

dbm

All database modules now support the `get()` and `setdefault()` methods.

(Suggested by Ray Allen in issue 9523.)

ctypes

A new type, `ctypes.c_ssize_t` represents the C `ssize_t` datatype.

site

The `site` module has three new functions useful for reporting on the details of a given Python installation.

- `getsitepackages()` lists all global site-packages directories.
- `getuserbase()` reports on the user's base directory where data can be stored.
- `getusersitepackages()` reveals the user-specific site-packages directory path.

```
>>> import site
>>> site.getsitepackages()
['/Library/Frameworks/Python.framework/Versions/3.2/lib/python3.2/site-pac
 '/Library/Frameworks/Python.framework/Versions/3.2/lib/site-python',
 '/Library/Python/3.2/site-packages']
>>> site.getuserbase()
'/Users/raymondhettinger/Library/Python/3.2'
>>> site.getusersitepackages()
'/Users/raymondhettinger/Library/Python/3.2/lib/python/site-packages'
```

Conveniently, some of site's functionality is accessible directly from the command-line:

```
$ python -m site --user-base
/Users/raymondhettinger/.local
$ python -m site --user-site
/Users/raymondhettinger/.local/lib/python3.2/site-packages
```

(Contributed by Tarek Ziadé in issue 6693.)

sysconfig

The new `sysconfig` module makes it straightforward to discover installation paths and configuration variables that vary across platforms and installations.

The module offers access simple access functions for platform and version information:

- `get_platform()` returning values like *linux-i586* or *macosx-10.6-ppc*.
- `get_python_version()` returns a Python version string such as "3.2".

It also provides access to the paths and variables corresponding to one of seven named schemes used by `distutils`. Those include *posix_prefix*, *posix_home*, *posix_user*, *nt*, *nt_user*, *os2*, *os2_home*:

- `get_paths()` makes a dictionary containing installation paths for the current installation scheme.
- `get_config_vars()` returns a dictionary of platform specific variables.

There is also a convenient command-line interface:

```
C:\Python32>python -m sysconfig
Platform: "win32"
Python version: "3.2"
Current installation scheme: "nt"

Paths:
        data = "C:\Python32"
        include = "C:\Python32\Include"
        platinclude = "C:\Python32\Include"
        platlib = "C:\Python32\Lib\site-packages"
        platstdlib = "C:\Python32\Lib"
        purelib = "C:\Python32\Lib\site-packages"
        scripts = "C:\Python32\Scripts"
        stdlib = "C:\Python32\Lib"

Variables:
        BINDIR = "C:\Python32"
        BINLIBDEST = "C:\Python32\Lib"
        EXE = ".exe"
        INCLUDEPY = "C:\Python32\Include"
        LIBDEST = "C:\Python32\Lib"
        SO = ".pyd"
        VERSION = "32"
        abiflags = ""
        base = "C:\Python32"
        exec_prefix = "C:\Python32"
        platbase = "C:\Python32"
        prefix = "C:\Python32"
        projectbase = "C:\Python32"
        py_version = "3.2"
        py_version_nodot = "32"
        py_version_short = "3.2"
        srcdir = "C:\Python32"
        userbase = "C:\Documents and Settings\Raymond\Application Data\Pyt
```

(Moved out of Distutils by Tarek Ziadé.)

pdb

The `pdb` debugger module gained a number of usability improvements:

- `pdb.py` now has a `-c` option that executes commands as given in a `.pdbrc` script file.
- A `.pdbrc` script file can contain `continue` and `next` commands that continue debugging.
- The `Pdb` class constructor now accepts a *nosigint* argument.
- New commands: `l(list)`, `ll(long list)` and `source` for listing source code.
- New commands: `display` and `undisplay` for showing or hiding the value of an expression if it has changed.
- New command: `interact` for starting an interactive interpreter containing the global and local names found in the current scope.
- Breakpoints can be cleared by breakpoint number.

(Contributed by Georg Brandl, Antonio Cuni and Ilya Sandler.)

configparser

The `configparser` module was modified to improve usability and predictability of the default parser and its supported INI syntax. The old `ConfigParser` class was removed in favor of `SafeConfigParser` which has in turn been renamed to `ConfigParser`. Support for inline comments is now turned off by default and section or option duplicates are not allowed in a single configuration source.

Config parsers gained a new API based on the mapping protocol:

```
>>> parser = ConfigParser()
>>> parser.read_string("""
[DEFAULT]
location = upper left
visible = yes
editable = no
color = blue

[main]
title = Main Menu
color = green

[options]
title = Options
""")
>>> parser['main']['color']
'green'
>>> parser['main']['editable']
'no'
>>> section = parser['options']
>>> section['title']
'Options'
>>> section['title'] = 'Options (editable: %(editable)s)'
>>> section['title']
'Options (editable: no)'
```

The new API is implemented on top of the classical API, so custom parser subclasses should be able to use it without modifications.

The INI file structure accepted by config parsers can now be customized. Users can specify alternative option/value delimiters and comment prefixes, change the name of the *DEFAULT* section or switch the interpolation syntax.

There is support for pluggable interpolation including an additional interpolation handler `ExtendedInterpolation`:

```
>>> parser = ConfigParser(interpolation=ExtendedInterpolation())
>>> parser.read_dict({'buildout': {'directory': '/home/ambv/zope9'},
                      'custom': {'prefix': '/usr/local'}})
>>> parser.read_string("""
    [buildout]
    parts =
      zope9
      instance
    find-links =
      ${buildout:directory}/downloads/dist

    [zope9]
    recipe = plone.recipe.zope9install
    location = /opt/zope

    [instance]
    recipe = plone.recipe.zope9instance
    zope9-location = ${zope9:location}
    zope-conf = ${custom:prefix}/etc/zope.conf
    """)
>>> parser['buildout']['find-links']
'\n/home/ambv/zope9/downloads/dist'
>>> parser['instance']['zope-conf']
'/usr/local/etc/zope.conf'
>>> instance = parser['instance']
>>> instance['zope-conf']
'/usr/local/etc/zope.conf'
>>> instance['zope9-location']
'/opt/zope'
```

A number of smaller features were also introduced, like support for specifying encoding in read operations, specifying fallback values for get-functions, or reading directly from dictionaries and strings.

(All changes contributed by Łukasz Langa.)

urllib.parse

A number of usability improvements were made for the `urllib.parse` module.

The `urlparse()` function now supports IPv6 addresses as described in **RFC 2732**:

```
>>> import urllib.parse
>>> urllib.parse.urlparse('http://[dead:beef:cafe:5417:affe:8FA3:deaf:feed
ParseResult(scheme='http',
            netloc='[dead:beef:cafe:5417:affe:8FA3:deaf:feed]',
            path='/foo/',
            params='',
            query='',
            fragment='')
```

The `urldefrag()` function now returns a *named tuple*:

```
>>> r = urllib.parse.urldefrag('http://python.org/about/#target')
>>> r
DefragResult(url='http://python.org/about/', fragment='target')
>>> r[0]
'http://python.org/about/'
>>> r.fragment
'target'
```

And, the `urlencode()` function is now much more flexible, accepting either a string or bytes type for the *query* argument. If it is a string, then the *safe*, *encoding*, and *error* parameters are sent to `quote_plus()` for encoding:

```
>>> urllib.parse.urlencode([
         ('type', 'telenovela'),
         ('name', '¿Dónde Está Elisa?')],
         encoding='latin-1')
'type=telenovela&name=%BFD%F3nde+Est%E1+Elisa%3F'
```

As detailed in *Parsing ASCII Encoded Bytes*, all the `urllib.parse` functions now accept ASCII-encoded byte strings as input, so long as they are not mixed with regular strings. If ASCII-encoded byte strings are given as parameters, the return types will also be an ASCII-encoded byte strings:

```
>>> urllib.parse.urlparse(b'http://www.python.org:80/about/')
ParseResultBytes(scheme=b'http', netloc=b'www.python.org:80',
                 path=b'/about/', params=b'', query=b'', fragment=b'')
```

(Work by Nick Coghlan, Dan Mahn, and Senthil Kumaran in issue 2987, issue 5468, and issue 9873.)

mailbox

Thanks to a concerted effort by R. David Murray, the `mailbox` module has been fixed for Python 3.2. The challenge was that mailbox had been originally designed with a text interface, but email messages are best represented with `bytes` because various parts of a message may have different encodings.

The solution harnessed the `email` package's binary support for parsing arbitrary email messages. In addition, the solution required a number of API changes.

As expected, the `add()` method for `mailbox.Mailbox` objects now accepts binary input.

`StringIO` and text file input are deprecated. Also, string input will fail early if non-ASCII characters are used. Previously it would fail when the email was processed in a later step.

There is also support for binary output. The `get_file()` method now returns a file in the binary mode (where it used to incorrectly set the file to text-mode). There is also a new `get_bytes()` method that returns a `bytes` representation of a message corresponding to a given *key*.

It is still possible to get non-binary output using the old API's `get_string()` method, but that approach is not very useful. Instead, it is best to extract messages from a `Message` object or to load them from binary input.

(Contributed by R. David Murray, with efforts from Steffen Daode Nurpmeso and an initial patch by Victor Stinner in issue 9124.)

turtledemo

The demonstration code for the `turtle` module was moved from the *Demo* directory to main library. It includes over a dozen sample scripts with lively displays. Being on `sys.path`, it can now be run directly from the command-line:

```
$ python -m turtledemo
```

(Moved from the Demo directory by Alexander Belopolsky in issue 10199.)

Multi-threading

- The mechanism for serializing execution of concurrently running Python threads (generally known as the *GIL* or *Global Interpreter Lock*) has been rewritten. Among the objectives were more predictable switching intervals and reduced overhead due to lock contention and the number of ensuing system calls. The notion of a "check interval" to allow thread switches has been abandoned and replaced by an absolute duration expressed in seconds. This parameter is tunable through `sys.setswitchinterval()`. It currently defaults to 5 milliseconds.

Additional details about the implementation can be read from a python-dev mailing-list message (however, "priority requests" as exposed in this message have not been kept for inclusion).

(Contributed by Antoine Pitrou.)

- Regular and recursive locks now accept an optional *timeout* argument to their `acquire()` method. (Contributed by Antoine Pitrou; issue 7316.)
- Similarly, `threading.Semaphore.acquire()` also gained a *timeout* argument. (Contributed by Torsten Landschoff; issue 850728.)
- Regular and recursive lock acquisitions can now be interrupted by signals on platforms using Pthreads. This means that Python programs that deadlock while acquiring locks can be successfully killed by repeatedly sending SIGINT to the process (by pressing `Ctrl+C` in most shells). (Contributed by Reid Kleckner; issue 8844.)

Optimizations

A number of small performance enhancements have been added:

- Python's peephole optimizer now recognizes patterns such `x in {1, 2, 3}` as being a test for membership in a set of constants. The optimizer recasts the `set` as a `frozenset` and stores the pre-built constant.

  Now that the speed penalty is gone, it is practical to start writing membership tests using set-notation. This style is both semantically clear and operationally fast:

  ```
  extension = name.rpartition('.')[2]
  if extension in {'xml', 'html', 'xhtml', 'css'}:
      handle(name)
  ```

  (Patch and additional tests contributed by Dave Malcolm; issue 6690).

- Serializing and unserializing data using the `pickle` module is now several times faster.

  (Contributed by Alexandre Vassalotti, Antoine Pitrou and the Unladen Swallow team in issue 9410 and issue 3873.)

- The Timsort algorithm used in `list.sort()` and `sorted()` now runs faster and uses less memory when called with a *key function*. Previously, every element of a list was wrapped with a temporary object that remembered the key value associated with each element. Now, two arrays of keys and values are sorted in parallel. This saves the memory consumed by the sort wrappers, and it saves time lost to delegating comparisons.

  (Patch by Daniel Stutzbach in issue 9915.)

- JSON decoding performance is improved and memory consumption is reduced whenever the same string is repeated for multiple keys. Also, JSON encoding now uses the C speedups when the `sort_keys` argument is true.

  (Contributed by Antoine Pitrou in issue 7451 and by Raymond Hettinger and Antoine Pitrou in issue 10314.)

- Recursive locks (created with the `threading.RLock()` API) now benefit from a C implementation which makes them as fast as regular locks, and between 10x and 15x faster than their previous pure Python implementation.

  (Contributed by Antoine Pitrou; issue 3001.)

- The fast-search algorithm in stringlib is now used by the `split()`, `splitlines()` and `replace()` methods on `bytes`, `bytearray` and `str` objects. Likewise, the algorithm is also used by `rfind()`, `rindex()`, `rsplit()` and `rpartition()`.

  (Patch by Florent Xicluna in issue 7622 and issue 7462.)

- Integer to string conversions now work two "digits" at a time, reducing the number of division and modulo operations.

  (issue 6713 by Gawain Bolton, Mark Dickinson, and Victor Stinner.)

There were several other minor optimizations. Set differencing now runs faster when one operand is much larger than the other (patch by Andress Bennetts in issue 8685). The `array.repeat()` method has a faster implementation (issue 1569291 by Alexander Belopolsky). The `BaseHTTPRequestHandler` has more efficient buffering (issue 3709 by Andrew Schaaf). The `operator.attrgetter()` function has been sped-up (issue 10160 by Christos Georgiou). And `ConfigParser` loads multi-line arguments a bit faster (issue 7113 by Łukasz Langa).

Unicode

Python has been updated to Unicode 6.0.0. The update to the standard adds over 2,000 new characters including emoji symbols which are important for mobile phones.

In addition, the updated standard has altered the character properties for two Kannada characters (U+0CF1, U+0CF2) and one New Tai Lue numeric character (U+19DA), making the former eligible for use in identifiers while disqualifying the latter. For more information, see Unicode Character Database Changes.

Codecs

Support was added for *cp720* Arabic DOS encoding (issue 1616979).

MBCS encoding no longer ignores the error handler argument. In the default strict mode, it raises an `UnicodeDecodeError` when it encounters an undecodable byte sequence and an `UnicodeEncodeError` for an unencodable character.

The MBCS codec supports `'strict'` and `'ignore'` error handlers for decoding, and `'strict'` and `'replace'` for encoding.

To emulate Python3.1 MBCS encoding, select the `'ignore'` handler for decoding and the `'replace'` handler for encoding.

On Mac OS X, Python decodes command line arguments with `'utf-8'` rather than the locale encoding.

By default, `tarfile` uses `'utf-8'` encoding on Windows (instead of `'mbcs'`) and the `'surrogateescape'` error handler on all operating systems.

Documentation

The documentation continues to be improved.

- A table of quick links has been added to the top of lengthy sections such as *Built-in Functions*. In the case of `itertools`, the links are accompanied by tables of cheatsheet-style summaries to provide an overview and memory jog without having to read all of the docs.
- In some cases, the pure Python source code can be a helpful adjunct to the documentation, so now many modules now feature quick links to the latest version of the source code. For example, the `functools` module documentation has a quick link at the top labeled:

  > **Source code** Lib/functools.py.

  (Contributed by Raymond Hettinger; see rationale.)
- The docs now contain more examples and recipes. In particular, `re` module has an extensive section, *Regular Expression Examples*. Likewise, the `itertools` module continues to be updated with new *Itertools Recipes*.
- The `datetime` module now has an auxiliary implementation in pure Python. No functionality was changed. This just provides an easier-to-read alternate implementation.

  (Contributed by Alexander Belopolsky in issue 9528.)
- The unmaintained `Demo` directory has been removed. Some demos were integrated into the documentation, some were moved to the `Tools/demo` directory, and others were removed altogether.

  (Contributed by Georg Brandl in issue 7962.)

IDLE

- The format menu now has an option to clean source files by stripping trailing whitespace.

(Contributed by Raymond Hettinger; issue 5150.)

- IDLE on Mac OS X now works with both Carbon AquaTk and Cocoa AquaTk.

  (Contributed by Kevin Walzer, Ned Deily, and Ronald Oussoren; issue 6075.)

Code Repository

In addition to the existing Subversion code repository at http://svn.python.org there is now a Mercurial repository at http://hg.python.org/.

After the 3.2 release, there are plans to switch to Mercurial as the primary repository. This distributed version control system should make it easier for members of the community to create and share external changesets. See **PEP 385** for details.

To learn the new version control system, see the tutorial by Joel Spolsky or the Guide to Mercurial Workflows.

Build and C API Changes

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

- The *idle*, *pydoc* and *2to3* scripts are now installed with a version-specific suffix on `make altinstall` (issue 10679).

- The C functions that access the Unicode Database now accept and return characters from the full Unicode range, even on narrow unicode builds (Py_UNICODE_TOLOWER, Py_UNICODE_ISDECIMAL, and others). A visible difference in Python is that `unicodedata.numeric()` now returns the correct value for large code points, and `repr()` may consider more characters as printable.

  (Reported by Bupjoe Lee and fixed by Amaury Forgeot D'Arc; issue 5127.)

- Computed gotos are now enabled by default on supported compilers (which are detected by the configure script). They can still be disabled selectively by specifying `--without-computed-gotos`.

  (Contributed by Antoine Pitrou; issue 9203.)

- The option `--with-wctype-functions` was removed. The built-in unicode database is now used for all functions.

  (Contributed by Amaury Forgeot D'Arc; issue 9210.)

- Hash values are now values of a new type, `Py_hash_t`, which is defined to be the same size as a pointer. Previously they were of type long, which on some 64-bit operating systems is still only 32 bits long. As a result of this fix, `set` and `dict` can now hold more than `2**32` entries on builds with 64-bit pointers (previously, they could grow to that size but their performance degraded catastrophically).

  (Suggested by Raymond Hettinger and implemented by Benjamin Peterson; issue 9778.)

- A new macro `Py_VA_COPY` copies the state of the variable argument list. It is equivalent to C99 *va_copy* but available on all Python platforms (issue 2443).

- A new C API function `PySys_SetArgvEx()` allows an embedded interpreter to set `sys.argv` without also modifying `sys.path` (issue 5753).

- `PyEval_CallObject` is now only available in macro form. The function declaration, which was kept for backwards compatibility reasons, is now removed – the macro was introduced in 1997 (issue 8276).

- There is a new function `PyLong_AsLongLongAndOverflow()` which is analogous to `PyLong_AsLongAndOverflow()`. They both serve to convert Python `int` into a native fixed-width type while providing detection of cases where the conversion won't fit (issue 7767).

- The `PyUnicode_CompareWithASCIIString()` function now returns *not equal* if the Python string is *NUL* terminated.

- There is a new function `PyErr_NewExceptionWithDoc()` that is like `PyErr_NewException()` but allows a docstring to be specified. This lets C exceptions have the same self-documenting capabilities as their pure Python counterparts (issue 7033).

- When compiled with the `--with-valgrind` option, the pymalloc allocator will be automatically disabled when running under Valgrind. This gives improved memory leak detection when running under Valgrind, while taking advantage of pymalloc at other times (issue 2422).

- Removed the `O?` format from the *PyArg_Parse* functions. The format is no longer used and it had never been documented (issue 8837).

There were a number of other small changes to the C-API. See the Misc/NEWS file for a complete list.

Also, there were a number of updates to the Mac OS X build, see Mac/BuildScript/README.txt for details. For users running a 32/64-bit build, there is a known problem with the default Tcl/Tk on Mac OS X 10.6. Accordingly, we recommend installing an updated alternative such as ActiveState Tcl/Tk 8.5.9. See http://www.python.org/download/mac/tcltk/ for additional details.

Porting to Python 3.2

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

- The `configparser` module has a number of clean-ups. The major change is to replace the old `ConfigParser` class with long-standing preferred alternative `SafeConfigParser`. In addition there are a number of smaller incompatibilities:
  - The interpolation syntax is now validated on `get()` and `set()` operations. In the default interpolation scheme, only two tokens with percent signs are valid: `%(name)s` and `%%`, the latter being an escaped percent sign.
  - The `set()` and `add_section()` methods now verify that values are actual strings. Formerly, unsupported types could be introduced unintentionally.
  - Duplicate sections or options from a single source now raise either `DuplicateSectionError` or `DuplicateOptionError`. Formerly, duplicates would silently overwrite a previous entry.
  - Inline comments are now disabled by default so now the **;** character can be safely used in values.
  - Comments now can be indented. Consequently, for **;** or **#** to appear at the start of a line in multiline values, it has to be interpolated. This keeps comment prefix characters in values from being mistaken as comments.
  - `""` is now a valid value and is no longer automatically converted to an empty string. For empty strings, use `"option ="` in a line.
- The `nntplib` module was reworked extensively, meaning that its APIs are often

incompatible with the 3.1 APIs.

- `bytearray` objects can no longer be used as filenames; instead, they should be converted to `bytes`.

- The `array.tostring()` and `array.fromstring()` have been renamed to `array.tobytes()` and `array.frombytes()` for clarity. The old names have been deprecated. (See issue 8990.)

- `PyArg_Parse*()` functions:

  - "t#" format has been removed: use "s#" or "s*" instead
  - "w" and "w#" formats has been removed: use "w*" instead
- The `PyCObject` type, deprecated in 3.1, has been removed. To wrap opaque C pointers in Python objects, the `PyCapsule` API should be used instead; the new type has a well-defined interface for passing typing safety information and a less complicated signature for calling a destructor.

- The `sys.setfilesystemencoding()` function was removed because it had a flawed design.

- The `random.seed()` function and method now salt string seeds with an sha512 hash function. To access the previous version of *seed* in order to reproduce Python 3.1 sequences, set the *version* argument to *1*, `random.seed(s, version=1)`.

- The previously deprecated `string.maketrans()` function has been removed in favor of the static methods `bytes.maketrans()` and `bytearray.maketrans()`. This change solves the confusion around which types were supported by the `string` module. Now, `str`, `bytes`, and `bytearray` each have their own **maketrans** and **translate** methods with intermediate translation tables of the appropriate type.

  (Contributed by Georg Brandl; issue 5675.)

- The previously deprecated `contextlib.nested()` function has been removed in favor of a plain `with` statement which can accept multiple context managers. The latter technique is faster (because it is built-in), and it does a better job finalizing multiple context managers when one of them raises an exception:

```
with open('mylog.txt') as infile, open('a.out', 'w') as outfile:
    for line in infile:
        if '<critical>' in line:
            outfile.write(line)
```

(Contributed by Georg Brandl and Mattias Brändström; appspot issue 53094.)

- `struct.pack()` now only allows bytes for the `s` string pack code. Formerly, it would accept text arguments and implicitly encode them to bytes using UTF-8. This was problematic because it made assumptions about the correct encoding and because a variable-length encoding can fail when writing to fixed length segment of a structure.

  Code such as `struct.pack('<6sHHBBB', 'GIF87a', x, y)` should be rewritten with to use bytes instead of text, `struct.pack('<6sHHBBB', b'GIF87a', x, y)`.

  (Discovered by David Beazley and fixed by Victor Stinner; issue 10783.)

- The `xml.etree.ElementTree` class now raises an `xml.etree.ElementTree.ParseError` when a parse fails. Previously it raised a `xml.parsers.expat.ExpatError`.

- The new, longer `str()` value on floats may break doctests which rely on the old output format.

- In `subprocess.Popen`, the default value for *close_fds* is now `True` under Unix; under Windows, it is `True` if the three standard streams are set to `None`, `False` otherwise. Previously, *close_fds* was always `False` by default, which produced difficult to solve bugs or race conditions when open file descriptors would leak into the child process.

- Support for legacy HTTP 0.9 has been removed from `urllib.request` and `http.client`. Such support is still present on the server side (in `http.server`).

  (Contributed by Antoine Pitrou, issue 10711.)

- SSL sockets in timeout mode now raise `socket.timeout` when a timeout occurs, rather than a generic `SSLError`.

  (Contributed by Antoine Pitrou, issue 10272.)

- The misleading functions `PyEval_AcquireLock()` and `PyEval_ReleaseLock()` have been officially deprecated. The thread-state aware APIs (such as `PyEval_SaveThread()` and `PyEval_RestoreThread()`) should be used instead.
- Due to security risks, `asyncore.handle_accept()` has been deprecated, and a new function, `asyncore.handle_accepted()`, was added to replace it.

  (Contributed by Giampaolo Rodola in issue 6706.)
- Due to the new *GIL* implementation, `PyEval_InitThreads()` cannot be called before `Py_Initialize()` anymore.

What's New In Python 3.3

This article explains the new features in Python 3.3, compared to 3.2. Python 3.3 was released on September 29, 2012. For full details, see the changelog.

See also: **PEP 398** – Python 3.3 Release Schedule

Summary – Release highlights

New syntax features:

- New `yield from` expression for *generator delegation*.
- The `u'unicode'` syntax is accepted again for `str` objects.

New library modules:

- `faulthandler` (helps debugging low-level crashes)
- `ipaddress` (high-level objects representing IP addresses and masks)
- `lzma` (compress data using the XZ / LZMA algorithm)
- `unittest.mock` (replace parts of your system under test with mock objects)
- `venv` (Python *virtual environments*, as in the popular `virtualenv` package)

New built-in features:

- Reworked *I/O exception hierarchy*.

Implementation improvements:

- Rewritten *import machinery* based on `importlib`.
- More compact *unicode strings*.
- More compact *attribute dictionaries*.

Significantly Improved Library Modules:

- C Accelerator for the *decimal* module.
- Better unicode handling in the *email* module (*provisional*).

Security improvements:

- Hash randomization is switched on by default.

Please read on for a comprehensive list of user-facing changes.

PEP 405: Virtual Environments

Virtual environments help create separate Python setups while sharing a system-wide base install, for ease of maintenance. Virtual environments have their own set of private site packages (i.e. locally-installed libraries), and are optionally segregated from the system-wide site packages. Their concept and implementation are inspired by the popular `virtualenv` third-party package, but benefit from tighter integration with the interpreter core.

This PEP adds the `venv` module for programmatic access, and the *pyvenv* script for command-line access and administration. The Python interpreter checks for a `pyvenv.cfg`, file whose existence signals the base of a virtual environment's directory tree.

See also:

PEP 405 – Python Virtual Environments
PEP written by Carl Meyer; implementation by Carl Meyer and Vinay Sajip

PEP 420: Implicit Namespace Packages

Native support for package directories that don't require `__init__.py` marker files and can automatically span multiple path segments (inspired by various third party approaches to namespace packages, as described in **PEP 420**)

See also:

PEP 420 – Implicit Namespace Packages
PEP written by Eric V. Smith; implementation by Eric V. Smith and Barry Warsaw

PEP 3118: New memoryview implementation and buffer protocol documentation

The implementation of **PEP 3118** has been significantly improved.

The new memoryview implementation comprehensively fixes all ownership and lifetime issues of dynamically allocated fields in the Py_buffer struct that led to multiple crash reports. Additionally, several functions that crashed or returned incorrect results for non-contiguous or multi-dimensional input have been fixed.

The memoryview object now has a PEP-3118 compliant getbufferproc() that checks the consumer's request type. Many new features have been added, most of them work in full generality for non-contiguous arrays and arrays with suboffsets.

The documentation has been updated, clearly spelling out responsibilities for both exporters and consumers. Buffer request flags are grouped into basic and compound flags. The memory layout of non-contiguous and multi-dimensional NumPy-style arrays is explained.

Features

- All native single character format specifiers in struct module syntax (optionally prefixed with '@') are now supported.
- With some restrictions, the cast() method allows changing of format and shape of C-contiguous arrays.
- Multi-dimensional list representations are supported for any array type.
- Multi-dimensional comparisons are supported for any array type.
- One-dimensional memoryviews of hashable (read-only) types with formats B, b or c are now hashable. (Contributed by Antoine Pitrou in issue 13411)
- Arbitrary slicing of any 1-D arrays type is supported. For example, it is now possible to reverse a memoryview in O(1) by using a negative step.

API changes

- The maximum number of dimensions is officially limited to 64.
- The representation of empty shape, strides and suboffsets is now an empty tuple instead of None.
- Accessing a memoryview element with format 'B' (unsigned bytes) now returns an integer (in accordance with the struct module syntax). For returning a bytes object the view must be cast to 'c' first.
- memoryview comparisons now use the logical structure of the operands and compare all array elements by value. All format strings in struct module syntax are supported. Views with unrecognised format strings are still permitted, but will always compare as unequal, regardless of view contents.
- For further changes see Build and C API Changes and Porting C code.

(Contributed by Stefan Krah in issue 10181)

See also: **PEP 3118** – Revising the Buffer Protocol

PEP 393: Flexible String Representation

The Unicode string type is changed to support multiple internal representations, depending on the character with the largest Unicode ordinal (1, 2, or 4 bytes) in the represented string. This allows a space-efficient representation in common cases, but gives access to full UCS-4 on all systems. For compatibility with existing APIs, several representations may exist in parallel; over time, this compatibility should be phased out.

On the Python side, there should be no downside to this change.

On the C API side, PEP 393 is fully backward compatible. The legacy API should remain available at least five years. Applications using the legacy API will not fully benefit of the memory reduction, or – worse – may use a bit more memory, because Python may have to maintain two versions of each string (in the legacy format and in the new efficient storage).

Functionality

Changes introduced by **PEP 393** are the following:

- Python now always supports the full range of Unicode codepoints, including non-BMP ones (i.e. from `U+0000` to `U+10FFFF`). The distinction between narrow and wide builds no longer exists and Python now behaves like a wide build, even under Windows.
- With the death of narrow builds, the problems specific to narrow builds have also been fixed, for example:
  - `len()` now always returns 1 for non-BMP characters, so `len('\U0010FFFF') == 1`;
  - surrogate pairs are not recombined in string literals, so `'\uDBFF\uDFFF' != '\U0010FFFF'`;
  - indexing or slicing non-BMP characters returns the expected value, so `'\U0010FFFF'[0]` now returns `'\U0010FFFF'` and not `'\uDBFF'`;
  - all other functions in the standard library now correctly handle non-BMP codepoints.

- The value of `sys.maxunicode` is now always `1114111` (`0x10FFFF` in hexadecimal). The `PyUnicode_GetMax()` function still returns either `0xFFFF` or `0x10FFFF` for backward compatibility, and it should not be used with the new Unicode API (see issue 13054).
- The `./configure` flag `--with-wide-unicode` has been removed.

Performance and resource usage

The storage of Unicode strings now depends on the highest codepoint in the string:

- pure ASCII and Latin1 strings (`U+0000-U+00FF`) use 1 byte per codepoint;
- BMP strings (`U+0000-U+FFFF`) use 2 bytes per codepoint;
- non-BMP strings (`U+10000-U+10FFFF`) use 4 bytes per codepoint.

The net effect is that for most applications, memory usage of string storage should decrease significantly – especially compared to former wide unicode builds – as, in many cases, strings will be pure ASCII even in international contexts (because many strings store non-human language data, such as XML fragments, HTTP headers, JSON-encoded data, etc.). We also hope that it will, for the same reasons, increase CPU cache efficiency on non-trivial applications. The memory usage of Python 3.3 is two to three times smaller than Python 3.2, and a little bit better than Python 2.7, on a Django benchmark (see the PEP for details).

See also:

PEP 393 – Flexible String Representation
PEP written by Martin von Löwis; implementation by Torsten Becker and Martin von Löwis.

PEP 397: Python Launcher for Windows

The Python 3.3 Windows installer now includes a `py` launcher application that can be used to launch Python applications in a version independent fashion.

This launcher is invoked implicitly when double-clicking `*.py` files. If only a single Python version is installed on the system, that version will be used to run the file. If multiple versions are installed, the most recent version is used by default, but this can be overridden by including a Unix-style "shebang line" in the Python script.

The launcher can also be used explicitly from the command line as the `py` application. Running `py` follows the same version selection rules as implicitly launching scripts, but a more specific version can be selected by passing appropriate arguments (such as `-3` to request Python 3 when Python 2 is also installed, or `-2.6` to specifclly request an earlier Python version when a more recent version is installed).

In addition to the launcher, the Windows installer now includes an option to add the newly installed Python to the system PATH (contributed by Brian Curtin in issue 3561).

See also:

PEP 397 – Python Launcher for Windows
PEP written by Mark Hammond and Martin v. Löwis; implementation by Vinay Sajip.

Launcher documentation: *Python Launcher for Windows*

Installer PATH modification: *Finding the Python executable*

PEP 3151: Reworking the OS and IO exception hierarchy

The hierarchy of exceptions raised by operating system errors is now both simplified and finer-grained.

You don't have to worry anymore about choosing the appropriate exception type between `OSError`, `IOError`, `EnvironmentError`, `WindowsError`, `mmap.error`, `socket.error` or `select.error`. All these exception types are now only one: `OSError`. The other names are kept as aliases for compatibility reasons.

Also, it is now easier to catch a specific error condition. Instead of inspecting the `errno` attribute (or `args[0]`) for a particular constant from the `errno` module, you can catch the adequate `OSError` subclass. The available subclasses are the following:

- `BlockingIOError`
- `ChildProcessError`
- `ConnectionError`

- `FileExistsError`
- `FileNotFoundError`
- `InterruptedError`
- `IsADirectoryError`
- `NotADirectoryError`
- `PermissionError`
- `ProcessLookupError`
- `TimeoutError`

And the `ConnectionError` itself has finer-grained subclasses:

- `BrokenPipeError`
- `ConnectionAbortedError`
- `ConnectionRefusedError`
- `ConnectionResetError`

Thanks to the new exceptions, common usages of the `errno` can now be avoided. For example, the following code written for Python 3.2:

```
from errno import ENOENT, EACCES, EPERM

try:
    with open("document.txt") as f:
        content = f.read()
except IOError as err:
    if err.errno == ENOENT:
        print("document.txt file is missing")
    elif err.errno in (EACCES, EPERM):
        print("You are not allowed to read document.txt")
    else:
        raise
```

can now be written without the `errno` import and without manual inspection of exception attributes:

```
try:
    with open("document.txt") as f:
        content = f.read()
except FileNotFoundError:
    print("document.txt file is missing")
except PermissionError:
    print("You are not allowed to read document.txt")
```

See also:

PEP 3151 – Reworking the OS and IO Exception Hierarchy
PEP written and implemented by Antoine Pitrou

PEP 380: Syntax for Delegating to a Subgenerator

PEP 380 adds the `yield from` expression, allowing a *generator* to delegate part of its operations to another generator. This allows a section of code containing `yield` to be factored out and placed in another generator. Additionally, the subgenerator is allowed to return with a value, and the value is made available to the delegating generator.

While designed primarily for use in delegating to a subgenerator, the `yield from` expression actually allows delegation to arbitrary subiterators.

For simple iterators, `yield from iterable` is essentially just a shortened form of `for item in iterable: yield item`:

```
>>> def g(x):
...     yield from range(x, 0, -1)
...     yield from range(x)
...
>>> list(g(5))
[5, 4, 3, 2, 1, 0, 1, 2, 3, 4]
```

However, unlike an ordinary loop, `yield from` allows subgenerators to receive sent and thrown values directly from the calling scope, and return a final value to the outer generator:

```
>>> def accumulate():
...     tally = 0
...     while 1:
...         next = yield
...         if next is None:
...             return tally
...         tally += next
...
>>> def gather_tallies(tallies):
...     while 1:
...         tally = yield from accumulate()
...         tallies.append(tally)
...
>>> tallies = []
>>> acc = gather_tallies(tallies)
>>> next(acc) # Ensure the accumulator is ready to accept values
>>> for i in range(4):
...     acc.send(i)
...
>>> acc.send(None) # Finish the first tally
>>> for i in range(5):
...     acc.send(i)
...
>>> acc.send(None) # Finish the second tally
>>> tallies
[6, 10]
```

The main principle driving this change is to allow even generators that are designed to be used with the `send` and `throw` methods to be split into multiple subgenerators as easily as a single large function can be split into multiple subfunctions.

See also:

PEP 380 – Syntax for Delegating to a Subgenerator
PEP written by Greg Ewing; implementation by Greg Ewing, integrated into 3.3 by Renaud Blanch, Ryan Kelly and Nick Coghlan; documentation by Zbigniew Jędrzejewski-Szmek and Nick Coghlan

PEP 409: Suppressing exception context

PEP 409 introduces new syntax that allows the display of the chained exception context to be disabled. This allows cleaner error messages in applications that convert between exception types:

```
>>> class D:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr) from None
...
```

```
>>> D({}).x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x
```

Without the `from None` suffix to suppress the cause, the original exception would be displayed by default:

```
>>> class C:
...     def __init__(self, extra):
...         self._extra_attributes = extra
...     def __getattr__(self, attr):
...         try:
...             return self._extra_attributes[attr]
...         except KeyError:
...             raise AttributeError(attr)
...
>>> C({}).x
Traceback (most recent call last):
  File "<stdin>", line 6, in __getattr__
KeyError: 'x'

During handling of the above exception, another exception occurred:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 8, in __getattr__
AttributeError: x
```

No debugging capability is lost, as the original exception context remains available if needed (for example, if an intervening library has incorrectly suppressed valuable underlying details):

```
>>> try:
...     D({}).x
... except AttributeError as exc:
...     print(repr(exc.__context__))
...
KeyError('x',)
```

See also:

PEP 409 - Suppressing exception context

PEP written by Ethan Furman; implemented by Ethan Furman and Nick Coghlan.

PEP 414: Explicit Unicode literals

To ease the transition from Python 2 for Unicode aware Python applications that make heavy use of Unicode literals, Python 3.3 once again supports the "`u`" prefix for string literals. This prefix has no semantic significance in Python 3, it is provided solely to reduce the number of purely mechanical changes in migrating to Python 3, making it easier for developers to focus on the more significant semantic changes (such as the stricter default separation of binary and text data).

See also:

PEP 414 – Explicit Unicode literals
PEP written by Armin Ronacher.

PEP 3155: Qualified name for classes and functions

Functions and class objects have a new `__qualname__` attribute representing the "path" from the module top-level to their definition. For global functions and classes, this is the same as `__name__`. For other functions and classes, it provides better information about where they were actually defined, and how they might be accessible from the global scope.

Example with (non-bound) methods:

```
>>> class C:
...     def meth(self):
...         pass
>>> C.meth.__name__
'meth'
>>> C.meth.__qualname__
'C.meth'
```

Example with nested classes:

```
>>> class C:
...     class D:
...         def meth(self):
```

```
...             pass
...
>>> C.D.__name__
'D'
>>> C.D.__qualname__
'C.D'
>>> C.D.meth.__name__
'meth'
>>> C.D.meth.__qualname__
'C.D.meth'
```

Example with nested functions:

```
>>> def outer():
...     def inner():
...         pass
...     return inner
...
>>> outer().__name__
'inner'
>>> outer().__qualname__
'outer.<locals>.inner'
```

The string representation of those objects is also changed to include the new, more precise information:

```
>>> str(C.D)
"<class '__main__.C.D'>"
>>> str(C.D.meth)
'<function C.D.meth at 0x7f46b9fe31e0>'
```

See also:

PEP 3155 – Qualified name for classes and functions
PEP written and implemented by Antoine Pitrou.

PEP 412: Key-Sharing Dictionary

Dictionaries used for the storage of objects' attributes are now able to share part of their internal storage between each other (namely, the part which stores the keys and their respective hashes). This reduces the memory consumption of programs creating many instances of non-builtin types.

See also:

PEP 412 – Key-Sharing Dictionary
PEP written and implemented by Mark Shannon.

PEP 362: Function Signature Object

A new function `inspect.signature()` makes introspection of python callables easy and straightforward. A broad range of callables is supported: python functions, decorated or not, classes, and `functools.partial()` objects. New classes `inspect.Signature`, `inspect.Parameter` and `inspect.BoundArguments` hold information about the call signatures, such as, annotations, default values, parameters kinds, and bound arguments, which considerably simplifies writing decorators and any code that validates or amends calling signatures or arguments.

See also:

PEP 362: – Function Signature Object
PEP written by Brett Cannon, Yury Selivanov, Larry Hastings, Jiwon Seo; implemented by Yury Selivanov.

PEP 421: Adding sys.implementation

A new attribute on the `sys` module exposes details specific to the implementation of the currently running interpreter. The initial set of attributes on `sys.implementation` are `name`, `version`, `hexversion`, and `cache_tag`.

The intention of `sys.implementation` is to consolidate into one namespace the implementation-specific data used by the standard library. This allows different Python implementations to share a single standard library code base much more easily. In its initial state, `sys.implementation` holds only a small portion of the implementation-specific data. Over time that ratio will shift in order to make the standard library more portable.

One example of improved standard library portability is `cache_tag`. As of Python 3.3, `sys.implementation.cache_tag` is used by `importlib` to support **PEP 3147** compliance. Any Python implementation that uses `importlib` for its built-in import system may use `cache_tag` to control the caching behavior for modules.

SimpleNamespace

The implementation of `sys.implementation` also introduces a new type to Python: `types.SimpleNamespace`. In contrast to a mapping-based namespace, like `dict`, `SimpleNamespace` is attribute-based, like `object`. However, unlike `object`, `SimpleNamespace` instances are writable. This means that you can add, remove, and modify the namespace through normal attribute access.

See also:

PEP 421 – Adding sys.implementation
PEP written and implemented by Eric Snow.

Using importlib as the Implementation of Import

issue 2377 – Replace __import__ w/ importlib.__import__ issue 13959 – Re-implement parts of `imp` in pure Python issue 14605 – Make import machinery explicit issue 14646 – Require loaders set __loader__ and __package__

The `__import__()` function is now powered by `importlib.__import__()`. This work leads to the completion of "phase 2" of **PEP 302**. There are multiple benefits to this change. First, it has allowed for more of the machinery powering import to be exposed instead of being implicit and hidden within the C code. It also provides a single implementation for all Python VMs supporting Python 3.3 to use, helping to end any VM-specific deviations in import semantics. And finally it eases the maintenance of import, allowing for future growth to occur.

For the common user, there should be no visible change in semantics. For those whose code currently manipulates import or calls import programmatically, the code changes that might possibly be required are covered in the Porting Python code section of this document.

New APIs

One of the large benefits of this work is the exposure of what goes into making the import statement work. That means the various importers that were once implicit are now fully exposed as part of the `importlib` package.

The abstract base classes defined in `importlib.abc` have been expanded to properly delineate between *meta path finders* and *path entry finders* by introducing `importlib.abc.MetaPathFinder` and `importlib.abc.PathEntryFinder`, respectively. The old ABC of `importlib.abc.Finder` is now only provided for backwards-compatibility and does not enforce any method requirements.

In terms of finders, `importlib.machinery.FileFinder` exposes the mechanism used to search for source and bytecode files of a module. Previously this class was an implicit member of `sys.path_hooks`.

For loaders, the new abstract base class `importlib.abc.FileLoader` helps write a loader that uses the file system as the storage mechanism for a module's code. The loader for source files (`importlib.machinery.SourceFileLoader`), sourceless bytecode files (`importlib.machinery.SourcelessFileLoader`), and extension modules (`importlib.machinery.ExtensionFileLoader`) are now available for direct use.

`ImportError` now has `name` and `path` attributes which are set when there is relevant data to provide. The message for failed imports will also provide the full name of the module now instead of just the tail end of the module's name.

The `importlib.invalidate_caches()` function will now call the method with the same name on all finders cached in `sys.path_importer_cache` to help clean up any stored state as necessary.

Visible Changes

For potential required changes to code, see the Porting Python code section.

Beyond the expanse of what `importlib` now exposes, there are other visible changes to import. The biggest is that `sys.meta_path` and `sys.path_hooks` now store all of the meta path finders and path entry hooks used by import. Previously the finders were implicit and hidden within the C code of import instead of being directly exposed. This means that one can now easily remove or change the order of the various finders to fit one's needs.

Another change is that all modules have a `__loader__` attribute, storing the loader used to create the module. **PEP 302** has been updated to make this attribute mandatory for loaders to implement, so in the future once 3rd-party loaders have been updated people will be able to rely on the existence of the attribute. Until such time, though, import is setting the module post-load.

Loaders are also now expected to set the `__package__` attribute from **PEP 366**. Once again, import itself is already setting this on all loaders from `importlib` and import itself is setting the attribute post-load.

`None` is now inserted into `sys.path_importer_cache` when no finder can be found on `sys.path_hooks`. Since `imp.NullImporter` is not directly exposed on `sys.path_hooks` it could no longer be relied upon to always be available to use as a value representing no finder found.

All other changes relate to semantic changes which should be taken into consideration when updating code for Python 3.3, and thus should be read about in the Porting Python code section of this document.

(Implementation by Brett Cannon)

Other Language Changes

Some smaller changes made to the core Python language are:

- Added support for Unicode name aliases and named sequences. Both `unicodedata.lookup()` and `'\N{...}'` now resolve name aliases, and `unicodedata.lookup()` resolves named sequences too.

  (Contributed by Ezio Melotti in issue 12753)

- Unicode database updated to UCD version 6.1.0

- Equality comparisons on `range()` objects now return a result reflecting the equality of the underlying sequences generated by those range objects. (issue 13201)

- The `count()`, `find()`, `rfind()`, `index()` and `rindex()` methods of `bytes` and `bytearray` objects now accept an integer between 0 and 255 as their first argument.

(Contributed by Petri Lehtinen in issue 12170)

- The `rjust()`, `ljust()`, and `center()` methods of `bytes` and `bytearray` now accept a `bytearray` for the `fill` argument. (Contributed by Petri Lehtinen in issue 12380.)
- New methods have been added to `list` and `bytearray`: `copy()` and `clear()` (issue 10516). Consequently, `MutableSequence` now also defines a `clear()` method (issue 11388).
- Raw bytes literals can now be written `rb"..."` as well as `br"..."`.

  (Contributed by Antoine Pitrou in issue 13748.)
- `dict.setdefault()` now does only one lookup for the given key, making it atomic when used with built-in types.

  (Contributed by Filip Gruszczyński in issue 13521.)
- The error messages produced when a function call does not match the function signature have been significantly improved.

  (Contributed by Benjamin Peterson.)

A Finer-Grained Import Lock

Previous versions of CPython have always relied on a global import lock. This led to unexpected annoyances, such as deadlocks when importing a module would trigger code execution in a different thread as a side-effect. Clumsy workarounds were sometimes employed, such as the `PyImport_ImportModuleNoBlock()` C API function.

In Python 3.3, importing a module takes a per-module lock. This correctly serializes importation of a given module from multiple threads (preventing the exposure of incompletely initialized modules), while eliminating the aforementioned annoyances.

(Contributed by Antoine Pitrou in issue 9260.)

Builtin functions and types

- `open()` gets a new *opener* parameter: the underlying file descriptor for the file object is then obtained by calling *opener* with (*file*, *flags*). It can be used to use custom flags like `os.O_CLOEXEC` for example. The `'x'` mode was added: open for exclusive creation, failing if the file already exists.
- `print()`: added the *flush* keyword argument. If the *flush* keyword argument is true, the stream is forcibly flushed.
- `hash()`: hash randomization is enabled by default, see `object.__hash__()` and `PYTHONHASHSEED`.
- The `str` type gets a new `casefold()` method: return a casefolded copy of the string, casefolded strings may be used for caseless matching. For example, `'ß'.casefold()` returns `'ss'`.
- The sequence documentation has been substantially rewritten to better explain the binary/text sequence distinction and to provide specific documentation sections for the individual builtin sequence types (issue 4966)

New Modules

faulthandler

This new debug module `faulthandler` contains functions to dump Python tracebacks explicitly, on a fault (a crash like a segmentation fault), after a timeout, or on a user signal. Call `faulthandler.enable()` to install fault handlers for the `SIGSEGV`, `SIGFPE`, `SIGABRT`, `SIGBUS`, and `SIGILL` signals. You can also enable them at startup by setting the `PYTHONFAULTHANDLER` environment variable or by using *-X* `faulthandler` command line option.

Example of a segmentation fault on Linux:

```
$ python -q -X faulthandler
>>> import ctypes
>>> ctypes.string_at(0)
Fatal Python error: Segmentation fault

Current thread 0x00007fb899f39700:
  File "/home/python/cpython/Lib/ctypes/__init__.py", line 486 in string_a
  File "<stdin>", line 1 in <module>
Segmentation fault
```

ipaddress

The new `ipaddress` module provides tools for creating and manipulating objects representing IPv4 and IPv6 addresses, networks and interfaces (i.e. an IP address associated with a specific IP subnet).

(Contributed by Google and Peter Moody in **PEP 3144**)

lzma

The newly-added `lzma` module provides data compression and decompression using the LZMA algorithm, including support for the `.xz` and `.lzma` file formats.

(Contributed by Nadeem Vawda and Per Øyvind Karlsen in issue 6715)

Improved Modules

abc

Improved support for abstract base classes containing descriptors composed with abstract methods. The recommended approach to declaring abstract descriptors is now to provide `__isabstractmethod__` as a dynamically updated property. The built-in descriptors have been updated accordingly.

- `abc.abstractproperty` has been deprecated, use `property` with `abc.abstractmethod()` instead.
- `abc.abstractclassmethod` has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
- `abc.abstractstaticmethod` has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.

(Contributed by Darren Dale in issue 11610)

`abc.ABCMeta.register()` now returns the registered subclass, which means it can now be used as a class decorator (issue 10868).

array

The `array` module supports the `long long` type using `q` and `Q` type codes.

(Contributed by Oren Tirosh and Hirokazu Yamamoto in issue 1172711)

base64

ASCII-only Unicode strings are now accepted by the decoding functions of the `base64` modern interface. For example, `base64.b64decode('YWJj')` returns `b'abc'`. (Contributed by Catalin Iacob in issue 13641.)

binascii

In addition to the binary objects they normally accept, the `a2b_` functions now all also accept ASCII-only strings as input. (Contributed by Antoine Pitrou in issue 13637.)

bz2

The `bz2` module has been rewritten from scratch. In the process, several new features have been added:

- New `bz2.open()` function: open a bzip2-compressed file in binary or text mode.
- `bz2.BZ2File` can now read from and write to arbitrary file-like objects, by means of its constructor's *fileobj* argument.

  (Contributed by Nadeem Vawda in issue 5863)
- `bz2.BZ2File` and `bz2.decompress()` can now decompress multi-stream inputs (such as those produced by the **pbzip2** tool). `bz2.BZ2File` can now also be used to create this type of file, using the `'a'` (append) mode.

  (Contributed by Nir Aides in issue 1625)
- `bz2.BZ2File` now implements all of the `io.BufferedIOBase` API, except for the `detach()` and `truncate()` methods.

codecs

The `mbcs` codec has been rewritten to handle correctly `replace` and `ignore` error handlers on all Windows versions. The `mbcs` codec now supports all error handlers, instead of only `replace` to encode and `ignore` to decode.

A new Windows-only codec has been added: `cp65001` (issue 13216). It is the Windows code page 65001 (Windows UTF-8, `CP_UTF8`). For example, it is used by `sys.stdout` if the console output code page is set to cp65001 (e.g., using `chcp 65001` command).

Multibyte CJK decoders now resynchronize faster. They only ignore the first byte of an invalid byte sequence. For example, `b'\xff\n'.decode('gb2312', 'replace')` now returns a `\n` after the replacement character.

(issue 12016)

Incremental CJK codec encoders are no longer reset at each call to their encode() methods. For example:

```
$ ./python -q
>>> import codecs
>>> encoder = codecs.getincrementalencoder('hz')('strict')
>>> b''.join(encoder.encode(x) for x in '\u52ff\u65bd\u65bc\u4eba\u3002 By
b'~{NpJ)l6HK!#~} Bye.'
```

This example gives `b'~{Np~}~{J)~}~{l6~}~{HK~}~{!#~} Bye.'` with older Python versions.

(issue 12100)

The `unicode_internal` codec has been deprecated.

collections

Addition of a new `ChainMap` class to allow treating a number of mappings as a single unit. (Written by Raymond Hettinger for issue 11089, made public in issue 11297)

The abstract base classes have been moved in a new `collections.abc` module, to better differentiate between the abstract and the concrete collections classes. Aliases for ABCs are still present in the `collections` module to preserve existing imports. (issue 11085)

The `Counter` class now supports the unary `+` and `-` operators, as well as the in-place operators `+=`, `-=`, `|=`, and `&=`. (Contributed by Raymond Hettinger in issue 13121.)

contextlib

`ExitStack` now provides a solid foundation for programmatic manipulation of context managers and similar cleanup functionality. Unlike the previous `contextlib.nested` API (which was deprecated and removed), the new API is designed to work correctly regardless of whether context managers acquire their resources in their `__init__` method (for example, file objects) or in their `__enter__` method (for example, synchronisation objects from the `threading` module).

(issue 13585)

crypt

Addition of salt and modular crypt format (hashing method) and the `mksalt()` function to the `crypt` module.

(issue 10924)

curses

- If the `curses` module is linked to the ncursesw library, use Unicode functions when Unicode strings or characters are passed (e.g. `waddwstr()`), and bytes functions otherwise (e.g. `waddstr()`).
- Use the locale encoding instead of `utf-8` to encode Unicode strings.
- `curses.window` has a new `curses.window.encoding` attribute.
- The `curses.window` class has a new `get_wch()` method to get a wide character
- The `curses` module has a new `unget_wch()` function to push a wide character so the next `get_wch()` will return it

(Contributed by Iñigo Serna in issue 6755)

datetime

- Equality comparisons between naive and aware `datetime` instances now return `False` instead of raising `TypeError` (issue 15006).
- New `datetime.datetime.timestamp()` method: Return POSIX timestamp corresponding to the `datetime` instance.

- The `datetime.datetime.strftime()` method supports formatting years older than 1000.
- The `datetime.datetime.astimezone()` method can now be called without arguments to convert datetime instance to the system timezone.

decimal

issue 7652 – integrate fast native decimal arithmetic.
C-module and libmpdec written by Stefan Krah.

The new C version of the decimal module integrates the high speed libmpdec library for arbitrary precision correctly-rounded decimal floating point arithmetic. libmpdec conforms to IBM's General Decimal Arithmetic Specification.

Performance gains range from 10x for database applications to 100x for numerically intensive applications. These numbers are expected gains for standard precisions used in decimal floating point arithmetic. Since the precision is user configurable, the exact figures may vary. For example, in integer bignum arithmetic the differences can be significantly higher.

The following table is meant as an illustration. Benchmarks are available at http://www.bytereef.org/mpdecimal/quickstart.html.

	decimal.py	_decimal	speedup
pi	42.02s	0.345s	120x
telco	172.19s	5.68s	30x
psycopg	3.57s	0.29s	12x

Features

- The `FloatOperation` signal optionally enables stricter semantics for mixing floats and Decimals.
- If Python is compiled without threads, the C version automatically disables the expensive thread local context machinery. In this case, the variable `HAVE_THREADS` is set to `False`.

API changes

- The C module has the following context limits, depending on the machine architecture:

	32-bit	64-bit
`MAX_PREC`	`425000000`	`999999999999999999`
`MAX_EMAX`	`425000000`	`999999999999999999`
`MIN_EMIN`	`-425000000`	`-999999999999999999`

- In the context templates (`DefaultContext`, `BasicContext` and `ExtendedContext`) the magnitude of `Emax` and `Emin` has changed to `999999`.
- The `Decimal` constructor in decimal.py does not observe the context limits and converts values with arbitrary exponents or precision exactly. Since the C version has internal limits, the following scheme is used: If possible, values are converted exactly, otherwise `InvalidOperation` is raised and the result is NaN. In the latter case it is always possible to use `create_decimal()` in order to obtain a rounded or inexact value.
- The power function in decimal.py is always correctly-rounded. In the C version, it is defined in terms of the correctly-rounded `exp()` and `ln()` functions, but the final result is only "almost always correctly rounded".
- In the C version, the context dictionary containing the signals is a `MutableMapping`. For speed reasons, `flags` and `traps` always refer to the same `MutableMapping` that the context was initialized with. If a new signal dictionary is assigned, `flags` and `traps` are updated with the new values, but they do not reference the RHS dictionary.
- Pickling a `Context` produces a different output in order to have a common interchange format for the Python and C versions.
- The order of arguments in the `Context` constructor has been changed to match the order displayed by `repr()`.
- The `watchexp` parameter in the `quantize()` method is deprecated.

email

Policy Framework

The email package now has a `policy` framework. A `Policy` is an object with several methods and properties that control how the email package behaves. The primary policy for Python 3.3 is the `Compat32` policy, which provides backward compatibility with the email package in Python 3.2. A `policy` can be specified when an email message is parsed by a `parser`, or when a `Message` object is created, or when an email is serialized using a `generator`. Unless overridden, a policy passed to a `parser` is inherited by all the `Message` object and sub-objects created by the `parser`. By default a `generator` will use the policy of the `Message` object it is serializing. The default policy is `compat32`.

The minimum set of controls implemented by all `policy` objects are:

max_line_length	The maximum length, excluding the linesep character(s), individual lines may have when a `Message` is serialized. Defaults to 78.
linesep	The character used to separate individual lines when a `Message` is serialized. Defaults to `\n`.
cte_type	`7bit` or `8bit`. `8bit` applies only to a `Bytes generator`, and means that non-ASCII may be used where allowed by the protocol (or where it exists in the original input).
raise_on_defect	Causes a `parser` to raise error when defects are encountered instead of adding them to the `Message` object's `defects` list.

A new policy instance, with new settings, is created using the `clone()` method of policy objects. `clone` takes any of the above controls as keyword arguments. Any control not specified in the call retains its default value. Thus you can create a policy that uses `\r\n` linesep characters like this:

```
mypolicy = compat32.clone(linesep='\r\n')
```

Policies can be used to make the generation of messages in the format needed by your application simpler. Instead of having to remember to specify `linesep='\r\n'` in all the places you call a `generator`, you can specify it once, when you set the policy used by the `parser` or the `Message`, whichever your program uses to create `Message` objects. On the other hand, if you need to generate messages in multiple forms, you can still specify the parameters in the appropriate `generator` call. Or you can have custom policy instances for your different cases, and pass those in when you create the `generator`.

Provisional Policy with New Header API

While the policy framework is worthwhile all by itself, the main motivation for introducing it is to allow the creation of new policies that implement new features for the email package in a way that maintains backward compatibility for those who do not use the new policies. Because the new policies introduce a new API, we are releasing them in Python 3.3 as a *provisional policy*. Backwards incompatible changes (up to and including removal of the code) may occur if deemed necessary by the core developers.

The new policies are instances of `EmailPolicy`, and add the following additional controls:

refold_source	Controls whether or not headers parsed by a `parser` are refolded by the `generator`. It can be `none`, `long`, or `all`. The default is `long`, which means that source headers with a line longer than `max_line_length` get refolded. `none` means no line get refolded, and `all` means that all lines get refolded.
header_factory	A callable that take a `name` and `value` and produces a custom header object.

The `header_factory` is the key to the new features provided by the new policies. When one of the new policies is used, any header retrieved from a `Message` object is an object produced by the `header_factory`, and any time you set a header on a `Message` it becomes an object produced by `header_factory`. All such header objects have a `name` attribute equal to the header name. Address and Date headers have additional attributes that give you access to the parsed data of the header. This means you can now do things like this:

```
>>> m = Message(policy=SMTP)
>>> m['To'] = 'Éric <foo@example.com>'
>>> m['to']
'Éric <foo@example.com>'
>>> m['to'].addresses
```

```
(Address(display_name='Éric', username='foo', domain='example.com'),)
>>> m['to'].addresses[0].username
'foo'
>>> m['to'].addresses[0].display_name
'Éric'
>>> m['Date'] = email.utils.localtime()
>>> m['Date'].datetime
datetime.datetime(2012, 5, 25, 21, 39, 24, 465484, tzinfo=datetime.timezor
>>> m['Date']
'Fri, 25 May 2012 21:44:27 -0400'
>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
Date: Fri, 25 May 2012 21:44:27 -0400
```

You will note that the unicode display name is automatically encoded as `utf-8` when the message is serialized, but that when the header is accessed directly, you get the unicode version. This eliminates any need to deal with the `email.header` `decode_header()` or `make_header()` functions.

You can also create addresses from parts:

```
>>> m['cc'] = [Group('pals', [Address('Bob', 'bob', 'example.com'),
...                           Address('Sally', 'sally', 'example.com')]),
...            Address('Bonzo', addr_spec='bonz@laugh.com')]
>>> print(m)
To: =?utf-8?q?=C3=89ric?= <foo@example.com>
Date: Fri, 25 May 2012 21:44:27 -0400
cc: pals: Bob <bob@example.com>, Sally <sally@example.com>;, Bonzo <bonz@l
```

Decoding to unicode is done automatically:

```
>>> m2 = message_from_string(str(m))
>>> m2['to']
'Éric <foo@example.com>'
```

When you parse a message, you can use the `addresses` and `groups` attributes of the header objects to access the groups and individual addresses:

```
>>> m2['cc'].addresses
(Address(display_name='Bob', username='bob', domain='example.com'), Addres
>>> m2['cc'].groups
(Group(display_name='pals', addresses=(Address(display_name='Bob', usernan
```

In summary, if you use one of the new policies, header manipulation works the way it ought to: your application works with unicode strings, and the email package transparently encodes and decodes the unicode to and from the RFC standard Content Transfer Encodings.

Other API Changes

New `BytesHeaderParser`, added to the `parser` module to complement `HeaderParser` and complete the Bytes API.

New utility functions:

- `format_datetime()`: given a `datetime`, produce a string formatted for use in an email header.
- `parsedate_to_datetime()`: given a date string from an email header, convert it into an aware `datetime`, or a naive `datetime` if the offset is `-0000`.
- `localtime()`: With no argument, returns the current local time as an aware `datetime` using the local `timezone`. Given an aware `datetime`, converts it into an aware `datetime` using the local `timezone`.

ftplib

- `ftplib.FTP` now accepts a `source_address` keyword argument to specify the `(host, port)` to use as the source address in the bind call when creating the outgoing socket. (Contributed by Giampaolo Rodolà in issue 8594.)
- The `FTP_TLS` class now provides a new `ccc()` function to revert control channel back to plaintext. This can be useful to take advantage of firewalls that know how to handle NAT with non-secure FTP without opening fixed ports. (Contributed by Giampaolo Rodolà in issue 12139)
- Added `ftplib.FTP.mlsd()` method which provides a parsable directory listing format and deprecates `ftplib.FTP.nlst()` and `ftplib.FTP.dir()`. (Contributed by Giampaolo Rodolà in issue 11072)

functools

The `functools.lru_cache()` decorator now accepts a `typed` keyword argument (that defaults to `False` to ensure that it caches values of different types that compare equal in separate cache slots. (Contributed by Raymond Hettinger in issue 13227.)

gc

It is now possible to register callbacks invoked by the garbage collector before and after collection using the new `callbacks` list.

hmac

A new `compare_digest()` function has been added to prevent side channel attacks on digests through timing analysis. (Contributed by Nick Coghlan and Christian Heimes in issue 15061)

http

`http.server.BaseHTTPRequestHandler` now buffers the headers and writes them all at once when `end_headers()` is called. A new method `flush_headers()` can be used to directly manage when the accumlated headers are sent. (Contributed by Andrew Schaaf in issue 3709.)

`http.server` now produces valid `HTML 4.01 strict` output. (Contributed by Ezio Melotti in issue 13295.)

`http.client.HTTPResponse` now has a `readinto()` method, which means it can be used as a `io.RawIOBase` class. (Contributed by John Kuhn in issue 13464.)

html

`html.parser.HTMLParser` is now able to parse broken markup without raising errors, therefore the *strict* argument of the constructor and the `HTMLParseError` exception are now deprecated. The ability to parse broken markup is the result of a number of bug fixes that are also available on the latest bug fix releases of Python 2.7/3.2. (Contributed by Ezio Melotti in issue 15114, and issue 14538, issue 13993, issue 13960, issue 13358, issue 1745761, issue 755670, issue 13357, issue 12629, issue 1200313, issue 670664, issue 13273, issue 12888, issue 7311)

A new `html5` dictionary that maps HTML5 named character references to the equivalent Unicode character(s) (e.g. `html5['gt;'] == '>'`) has been added to the `html.entities` module. The dictionary is now also used by `HTMLParser`. (Contributed by Ezio Melotti in issue 11113 and issue 15156)

imaplib

The `IMAP4_SSL` constructor now accepts an SSLContext parameter to control parameters of the secure channel.

(Contributed by Sijin Joseph in issue 8808)

inspect

A new `getclosurevars()` function has been added. This function reports the current binding of all names referenced from the function body and where those names were resolved, making it easier to verify correct internal state when testing code that relies on stateful closures.

(Contributed by Meador Inge and Nick Coghlan in issue 13062)

A new `getgeneratorlocals()` function has been added. This function reports the current binding of local variables in the generator's stack frame, making it easier to verify correct internal state when testing generators.

(Contributed by Meador Inge in issue 15153)

io

The `open()` function has a new `'x'` mode that can be used to exclusively create a new file, and raise a `FileExistsError` if the file already exists. It is based on the C11 'x' mode to fopen().

(Contributed by David Townshend in issue 12760)

The constructor of the `TextIOWrapper` class has a new *write_through* optional argument. If *write_through* is `True`, calls to `write()` are guaranteed not to be buffered: any data written on the `TextIOWrapper` object is immediately handled to its underlying binary buffer.

itertools

`accumulate()` now takes an optional `func` argument for providing a user-supplied binary function.

logging

The `basicConfig()` function now supports an optional `handlers` argument taking an iterable of handlers to be added to the root logger.

A class level attribute `append_nul` has been added to `SysLogHandler` to allow control of the appending of the `NUL` (`\000`) byte to syslog records, since for some deamons it is required while for others it is passed through to the log.

math

The `math` module has a new function, `log2()`, which returns the base-2 logarithm of *x*.

(Written by Mark Dickinson in issue 11888).

mmap

The `read()` method is now more compatible with other file-like objects: if the argument is omitted or specified as `None`, it returns the bytes from the current file position to the end of the mapping. (Contributed by Petri Lehtinen in issue 12021.)

multiprocessing

The new `multiprocessing.connection.wait()` function allows to poll multiple objects (such as connections, sockets and pipes) with a timeout. (Contributed by Richard Oudkerk in issue 12328.)

`multiprocessing.Connection` objects can now be transferred over multiprocessing connections. (Contributed by Richard Oudkerk in issue 4892.)

`multiprocessing.Process` now accepts a `daemon` keyword argument to override the default behavior of inheriting the `daemon` flag from the parent process (issue 6064).

New attribute attribute `multiprocessing.Process.sentinel` allows a program to wait on multiple `Process` objects at one time using the appropriate OS primitives (for example, `select` on posix systems).

New methods `multiprocessing.pool.Pool.starmap()` and `starmap_async()` provide `itertools.starmap()` equivalents to the existing `multiprocessing.pool.Pool.map()` and `map_async()` functions. (Contributed by Hynek Schlawack in issue 12708.)

nntplib

The `nntplib.NNTP` class now supports the context manager protocol to unconditionally consume `socket.error` exceptions and to close the NNTP connection when done:

```
>>> from nntplib import NNTP
>>> with NNTP('news.gmane.org') as n:
...     n.group('gmane.comp.python.committers')
...
('211 1755 1 1755 gmane.comp.python.committers', 1755, 1, 1755, 'gmane.con
>>>
```

(Contributed by Giampaolo Rodolà in issue 9795)

os

- The `os` module has a new `pipe2()` function that makes it possible to create a pipe with `O_CLOEXEC` or `O_NONBLOCK` flags set atomically. This is especially useful to avoid race conditions in multi-threaded programs.

- The `os` module has a new `sendfile()` function which provides an efficent "zero-copy" way for copying data from one file (or socket) descriptor to another. The phrase "zero-copy" refers to the fact that all of the copying of data between the two descriptors is done entirely by the kernel, with no copying of data into userspace buffers. `sendfile()` can be used to efficiently copy data from a file on disk to a network socket, e.g. for downloading a file.

  (Patch submitted by Ross Lagerwall and Giampaolo Rodolà in issue 10882.)

- To avoid race conditions like symlink attacks and issues with temporary files and directories, it is more reliable (and also faster) to manipulate file descriptors instead of file names. Python 3.3 enhances existing functions and introduces new functions to work on file descriptors (issue 4761, issue 10755 and issue 14626).

  - The `os` module has a new `fwalk()` function similar to `walk()` except that it also yields file descriptors referring to the directories visited. This is especially useful to avoid symlink races.
  - The following functions get new optional *dir_fd* (*paths relative to directory descriptors*) and/or *follow_symlinks* (*not following symlinks*): `access()`, `chflags()`, `chmod()`, `chown()`, `link()`, `lstat()`, `mkdir()`, `mkfifo()`, `mknod()`, `open()`, `readlink()`, `remove()`, `rename()`, `replace()`, `rmdir()`, `stat()`, `symlink()`, `unlink()`, `utime()`. Platform support for using these parameters can be checked via the sets `os.supports_dir_fd` and `os.supports_follows_symlinks`.
  - The following functions now support a file descriptor for their path argument: `chdir()`, `chmod()`, `chown()`, `execve()`, `listdir()`, `pathconf()`, `exists()`, `stat()`, `statvfs()`, `utime()`. Platform support for this can be checked via the `os.supports_fd` set.

- `access()` accepts an `effective_ids` keyword argument to turn on using the effective uid/gid rather than the real uid/gid in the access check. Platform support for this can be checked via the `supports_effective_ids` set.

- The `os` module has two new functions: `getpriority()` and `setpriority()`. They can be used to get or set process niceness/priority in a fashion similar to `os.nice()` but extended to all processes instead of just the current one.

  (Patch submitted by Giampaolo Rodolà in issue 10784.)

- The new `os.replace()` function allows cross-platform renaming of a file with overwriting the destination. With `os.rename()`, an existing destination file is overwritten under POSIX, but raises an error under Windows. (Contributed by Antoine Pitrou in issue 8828.)

- The stat family of functions (`stat()`, `fstat()`, and `lstat()`) now support reading a file's timestamps with nanosecond precision. Symmetrically, `utime()` can now write file timestamps with nanosecond precision. (Contributed by Larry Hastings in issue 14127.)

- The new `os.get_terminal_size()` function queries the size of the terminal attached to a file descriptor. See also `shutil.get_terminal_size()`. (Contributed by Zbigniew Jędrzejewski-Szmek in issue 13609.)

- New functions to support Linux extended attributes (issue 12720): `getxattr()`, `listxattr()`, `removexattr()`, `setxattr()`.
- New interface to the scheduler. These functions control how a process is allocated CPU time by the operating system. New functions: `sched_get_priority_max()`, `sched_get_priority_min()`, `sched_getaffinity()`, `sched_getparam()`, `sched_getscheduler()`, `sched_rr_get_interval()`, `sched_setaffinity()`, `sched_setparam()`, `sched_setscheduler()`, `sched_yield()`,
- New functions to control the file system:
  - `posix_fadvise()`: Announces an intention to access data in a specific pattern thus allowing the kernel to make optimizations.
  - `posix_fallocate()`: Ensures that enough disk space is allocated for a file.
  - `sync()`: Force write of everything to disk.
- Additional new posix functions:
  - `lockf()`: Apply, test or remove a POSIX lock on an open file descriptor.
  - `pread()`: Read from a file descriptor at an offset, the file offset remains unchanged.
  - `pwrite()`: Write to a file descriptor from an offset, leaving the file offset unchanged.
  - `readv()`: Read from a file descriptor into a number of writable buffers.
  - `truncate()`: Truncate the file corresponding to *path*, so that it is at most *length* bytes in size.
  - `waitid()`: Wait for the completion of one or more child processes.
  - `writev()`: Write the contents of *buffers* to a file descriptor, where *buffers* is an arbitrary sequence of buffers.
  - `getgrouplist()` (issue 9344): Return list of group ids that specified user belongs to.
- `times()` and `uname()`: Return type changed from a tuple to a tuple-like object with named attributes.
- Some platforms now support additional constants for the `lseek()` function, such as `os.SEEK_HOLE` and `os.SEEK_DATA`.
- New constants `RTLD_LAZY`, `RTLD_NOW`, `RTLD_GLOBAL`, `RTLD_LOCAL`, `RTLD_NODELETE`, `RTLD_NOLOAD`, and `RTLD_DEEPBIND` are available on platforms that support them. These are for use with the `sys.setdlopenflags()` function, and supersede the similar constants defined in `ctypes` and `DLFCN`. (Contributed by Victor Stinner in issue 13226.)
- `os.symlink()` now accepts (and ignores) the `target_is_directory` keyword

argument on non-Windows platforms, to ease cross-platform support.

pdb

Tab-completion is now available not only for command names, but also their arguments. For example, for the `break` command, function and file names are completed.

(Contributed by Georg Brandl in issue 14210)

pickle

`pickle.Pickler` objects now have an optional `dispatch_table` attribute allowing to set per-pickler reduction functions.

(Contributed by Richard Oudkerk in issue 14166.)

pydoc

The Tk GUI and the `serve()` function have been removed from the `pydoc` module: `pydoc -g` and `serve()` have been deprecated in Python 3.2.

re

`str` regular expressions now support `\u` and `\U` escapes.

(Contributed by Serhiy Storchaka in issue 3665.)

sched

- `run()` now accepts a *blocking* parameter which when set to False makes the method execute the scheduled events due to expire soonest (if any) and then return immediately. This is useful in case you want to use the `scheduler` in non-blocking applications. (Contributed by Giampaolo Rodolà in issue 13449)
- `scheduler` class can now be safely used in multi-threaded environments. (Contributed by Josiah Carlson and Giampaolo Rodolà in issue 8684)
- *timefunc* and *delayfunct* parameters of `scheduler` class constructor are now optional and defaults to `time.time()` and `time.sleep()` respectively.

(Contributed by Chris Clark in issue 13245)

- `enter()` and `enterabs()` *argument* parameter is now optional. (Contributed by Chris Clark in issue 13245)
- `enter()` and `enterabs()` now accept a *kwargs* parameter. (Contributed by Chris Clark in issue 13245)

select

Solaris and derivative platforms have a new class `select.devpoll` for high performance asynchronous sockets via `/dev/poll`. (Contributed by Jesús Cea Avión in issue 6397.)

shlex

The previously undocumented helper function `quote` from the `pipes` modules has been moved to the `shlex` module and documented. `quote()` properly escapes all characters in a string that might be otherwise given special meaning by the shell.

shutil

- New functions:
  - `disk_usage()`: provides total, used and free disk space statistics. (Contributed by Giampaolo Rodolà in issue 12442)
  - `chown()`: allows one to change user and/or group of the given path also specifying the user/group names and not only their numeric ids. (Contributed by Sandro Tosi in issue 12191)
  - `shutil.get_terminal_size()`: returns the size of the terminal window to which the interpreter is attached. (Contributed by Zbigniew Jędrzejewski-Szmek in issue 13609.)
- `copy2()` and `copystat()` now preserve file timestamps with nanosecond precision on platforms that support it. They also preserve file "extended attributes" on Linux. (Contributed by Larry Hastings in issue 14127 and issue 15238.)
- Several functions now take an optional `symlinks` argument: when that parameter is true, symlinks aren't dereferenced and the operation instead acts on the symlink itself (or creates one, if relevant). (Contributed by Hynek Schlawack in issue 12715.)
- When copying files to a different file system, `move()` now handles symlinks the way the posix `mv` command does, recreating the symlink rather than copying the

target file contents. (Contributed by Jonathan Niehof in issue 9993.) `move()` now also returns the `dst` argument as its result.
- `rmtree()` is now resistant to symlink attacks on platforms which support the new `dir_fd` parameter in `os.open()` and `os.unlink()`. (Contributed by Martin von Löwis and Hynek Schlawack in issue 4489.)

signal

- The `signal` module has new functions:
  - `pthread_sigmask()`: fetch and/or change the signal mask of the calling thread (Contributed by Jean-Paul Calderone in issue 8407);
  - `pthread_kill()`: send a signal to a thread;
  - `sigpending()`: examine pending functions;
  - `sigwait()`: wait a signal;
  - `sigwaitinfo()`: wait for a signal, returning detailed information about it;
  - `sigtimedwait()`: like `sigwaitinfo()` but with a timeout.
- The signal handler writes the signal number as a single byte instead of a nul byte into the wakeup file descriptor. So it is possible to wait more than one signal and know which signals were raised.
- `signal.signal()` and `signal.siginterrupt()` raise an OSError, instead of a RuntimeError: OSError has an errno attribute.

smtpd

The `smtpd` module now supports **RFC 5321** (extended SMTP) and **RFC 1870** (size extension). Per the standard, these extensions are enabled if and only if the client initiates the session with an `EHLO` command.

(Initial `ELHO` support by Alberto Trevino. Size extension by Juhana Jauhiainen. Substantial additional work on the patch contributed by Michele Orrù and Dan Boswell. issue 8739)

smtplib

The `SMTP`, `SMTP_SSL`, and `LMTP` classes now accept a `source_address` keyword argument to specify the `(host, port)` to use as the source address in the bind call when creating the outgoing socket. (Contributed by Paulo Scardine in issue 11281.)

`SMTP` now supports the context manager protocol, allowing an `SMTP` instance to be used in a `with` statement. (Contributed by Giampaolo Rodolà in issue 11289.)

The `SMTP_SSL` constructor and the `starttls()` method now accept an SSLContext parameter to control parameters of the secure channel. (Contributed by Kasun Herath in issue 8809)

socket

- The `socket` class now exposes additional methods to process ancillary data when supported by the underlying platform:
    - `sendmsg()`
    - `recvmsg()`
    - `recvmsg_into()`

  (Contributed by David Watson in issue 6560, based on an earlier patch by Heiko Wundram)

- The `socket` class now supports the PF_CAN protocol family (http://en.wikipedia.org/wiki/Socketcan), on Linux (http://lwn.net/Articles/253425).

  (Contributed by Matthias Fuchs, updated by Tiago Gonçalves in issue 10141)

- The `socket` class now supports the PF_RDS protocol family (http://en.wikipedia.org/wiki/Reliable_Datagram_Sockets and http://oss.oracle.com/projects/rds/).

- The `socket` class now supports the `PF_SYSTEM` protocol family on OS X. (Contributed by Michael Goderbauer in issue 13777.)

- New function `sethostname()` allows the hostname to be set on unix systems if the calling process has sufficient privileges. (Contributed by Ross Lagerwall in issue 10866.)

socketserver

`BaseServer` now has an overridable method `service_actions()` that is called by the `serve_forever()` method in the service loop. `ForkingMixIn` now uses this to clean up zombie child proceses. (Contributed by Justin Warkentin in issue 11109.)

sqlite3

New `sqlite3.Connection` method `set_trace_callback()` can be used to capture a trace of all sql commands processed by sqlite. (Contributed by Torsten Landschoff in issue 11688.)

ssl

- The `ssl` module has two new random generation functions:
  - `RAND_bytes()`: generate cryptographically strong pseudo-random bytes.
  - `RAND_pseudo_bytes()`: generate pseudo-random bytes.

  (Contributed by Victor Stinner in issue 12049)
- The `ssl` module now exposes a finer-grained exception hierarchy in order to make it easier to inspect the various kinds of errors. (Contributed by Antoine Pitrou in issue 11183)
- `load_cert_chain()` now accepts a *password* argument to be used if the private key is encrypted. (Contributed by Adam Simpkins in issue 12803)
- Diffie-Hellman key exchange, both regular and Elliptic Curve-based, is now supported through the `load_dh_params()` and `set_ecdh_curve()` methods. (Contributed by Antoine Pitrou in issue 13626 and issue 13627)
- SSL sockets have a new `get_channel_binding()` method allowing the implementation of certain authentication mechanisms such as SCRAM-SHA-1-PLUS. (Contributed by Jacek Konieczny in issue 12551)
- You can query the SSL compression algorithm used by an SSL socket, thanks to its new `compression()` method. The new attribute `OP_NO_COMPRESSION` can be used to disable compression. (Contributed by Antoine Pitrou in issue 13634)
- Support has been added for the Next Procotol Negotiation extension using the `ssl.SSLContext.set_npn_protocols()` method. (Contributed by Colin Marc in

issue 14204)

- SSL errors can now be introspected more easily thanks to `library` and `reason` attributes. (Contributed by Antoine Pitrou in issue 14837)
- The `get_server_certificate()` function now supports IPv6. (Contributed by Charles-François Natali in issue 11811.)
- New attribute `OP_CIPHER_SERVER_PREFERENCE` allows setting SSLv3 server sockets to use the server's cipher ordering preference rather than the client's (issue 13635).

stat

The undocumented tarfile.filemode function has been moved to `stat.filemode()`. It can be used to convert a file's mode to a string of the form '-rwxrwxrwx'.

(Contributed by Giampaolo Rodolà in issue 14807)

struct

The `struct` module now supports `ssize_t` and `size_t` via the new codes `n` and `N`, respectively. (Contributed by Antoine Pitrou in issue 3163.)

subprocess

Command strings can now be bytes objects on posix platforms. (Contributed by Victor Stinner in issue 8513.)

A new constant `DEVNULL` allows suppressing output in a platform-independent fashion. (Contributed by Ross Lagerwall in issue 5870.)

sys

The `sys` module has a new `thread_info` *struct sequence* holding informations about the thread implementation (issue 11223).

tarfile

`tarfile` now supports `lzma` encoding via the `lzma` module. (Contributed by Lars Gustäbel in issue 5689.)

tempfile

`tempfile.SpooledTemporaryFile`'s `truncate()` method now accepts a `size` parameter. (Contributed by Ryan Kelly in issue 9957.)

textwrap

The `textwrap` module has a new `indent()` that makes it straightforward to add a common prefix to selected lines in a block of text (issue 13857).

threading

`threading.Condition`, `threading.Semaphore`, `threading.BoundedSemaphore`, `threading.Event`, and `threading.Timer`, all of which used to be factory functions returning a class instance, are now classes and may be subclassed. (Contributed by Éric Araujo in issue 10968).

The `threading.Thread` constructor now accepts a `daemon` keyword argument to override the default behavior of inheriting the `deamon` flag value from the parent thread (issue 6064).

The formerly private function `_thread.get_ident` is now available as the public function `threading.get_ident()`. This eliminates several cases of direct access to the `_thread` module in the stdlib. Third party code that used `_thread.get_ident` should likewise be changed to use the new public interface.

time

The **PEP 418** added new functions to the `time` module:

- `get_clock_info()`: Get information on a clock.
- `monotonic()`: Monotonic clock (cannot go backward), not affected by system clock updates.
- `perf_counter()`: Performance counter with the highest available resolution to measure a short duration.

- `process_time()`: Sum of the system and user CPU time of the current process.

Other new functions:

- `clock_getres()`, `clock_gettime()` and `clock_settime()` functions with `CLOCK_xxx` constants. (Contributed by Victor Stinner in issue 10278)

To improve cross platform consistency, `sleep()` now raises a `ValueError` when passed a negative sleep value. Previously this was an error on posix, but produced an infinite sleep on Windows.

types

Add a new `types.MappingProxyType` class: Read-only proxy of a mapping. (issue 14386)

The new functions *types.new_class* and *types.prepare_class* provide support for PEP 3115 compliant dynamic type creation. (issue 14588)

unittest

`assertRaises()`, `assertRaisesRegex()`, `assertWarns()`, and `assertWarnsRegex()` now accept a keyword argument *msg* when used as context managers. (Contributed by Ezio Melotti and Winston Ewert in issue 10775)

`unittest.TestCase.run()` now returns the `TestResult` object.

urllib

The `Request` class, now accepts a *method* argument used by `get_method()` to determine what HTTP method should be used. For example, this will send a `'HEAD'` request:

```
>>> urlopen(Request('http://www.python.org', method='HEAD'))
```

(issue 1673007)

webbrowser

The `webbrowser` module supports more "browsers": Google Chrome (named **chrome**, **chromium**, **chrome-browser** or **chromium-browser** depending on the version and operating system), and the generic launchers **xdg-open**, from the FreeDesktop.org project, and **gvfs-open**, which is the default URI handler for GNOME 3. (The former contributed by Arnaud Calmettes in issue 13620, the latter by Matthias Klose in issue 14493)

xml.etree.ElementTree

The `xml.etree.ElementTree` module now imports its C accelerator by default; there is no longer a need to explicitly import `xml.etree.cElementTree` (this module stays for backwards compatibility, but is now deprecated). In addition, the `iter` family of methods of `Element` has been optimized (rewritten in C). The module's documentation has also been greatly improved with added examples and a more detailed reference.

zlib

New attribute `zlib.Decompress.eof` makes it possible to distinguish between a properly-formed compressed stream and an incomplete or truncated one. (Contributed by Nadeem Vawda in issue 12646.)

New attribute `zlib.ZLIB_RUNTIME_VERSION` reports the version string of the underlying `zlib` library that is loaded at runtime. (Contributed by Torsten Landschoff in issue 12306.)

Optimizations

Major performance enhancements have been added:

- Thanks to **PEP 393**, some operations on Unicode strings have been optimized:
    - the memory footprint is divided by 2 to 4 depending on the text
    - encode an ASCII string to UTF-8 doesn't need to encode characters anymore, the UTF-8 representation is shared with the ASCII representation
    - the UTF-8 encoder has been optimized
    - repeating a single ASCII letter and getting a substring of a ASCII strings is 4 times faster

- UTF-8 is now 2x to 4x faster. UTF-16 encoding is now up to 10x faster.

  (contributed by Serhiy Storchaka, issue 14624, issue 14738 and issue 15026.)

Build and C API Changes

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

- New **PEP 3118** related function:
  - `PyMemoryView_FromMemory()`
- **PEP 393** added new Unicode types, macros and functions:
  - High-level API:
    - `PyUnicode_CopyCharacters()`
    - `PyUnicode_FindChar()`
    - `PyUnicode_GetLength()`, `PyUnicode_GET_LENGTH`
    - `PyUnicode_New()`
    - `PyUnicode_Substring()`
    - `PyUnicode_ReadChar()`, `PyUnicode_WriteChar()`
  - Low-level API:
    - `Py_UCS1`, `Py_UCS2`, `Py_UCS4` types
    - `PyASCIIObject` and `PyCompactUnicodeObject` structures
    - `PyUnicode_READY`
    - `PyUnicode_FromKindAndData()`
    - `PyUnicode_AsUCS4()`, `PyUnicode_AsUCS4Copy()`
    - `PyUnicode_DATA`, `PyUnicode_1BYTE_DATA`, `PyUnicode_2BYTE_DATA`, `PyUnicode_4BYTE_DATA`
    - `PyUnicode_KIND` with `PyUnicode_Kind` enum: `PyUnicode_WCHAR_KIND`, `PyUnicode_1BYTE_KIND`, `PyUnicode_2BYTE_KIND`, `PyUnicode_4BYTE_KIND`
    - `PyUnicode_READ`, `PyUnicode_READ_CHAR`, `PyUnicode_WRITE`
    - `PyUnicode_MAX_CHAR_VALUE`
- `PyArg_ParseTuple` now accepts a `bytearray` for the `c` format (issue 12380).

Deprecated

Unsupported Operating Systems

OS/2 and VMS are no longer supported due to the lack of a maintainer.

Windows 2000 and Windows platforms which set `COMSPEC` to `command.com` are no longer supported due to maintenance burden.

OSF support, which was deprecated in 3.2, has been completely removed.

Deprecated Python modules, functions and methods

- Passing a non-empty string to `object.__format__()` is deprecated, and will produce a `TypeError` in Python 3.4 (issue 9856).
- The `unicode_internal` codec has been deprecated because of the **PEP 393**, use UTF-8, UTF-16 (`utf-16-le` or `utf-16-be`), or UTF-32 (`utf-32-le` or `utf-32-be`)
- `ftplib.FTP.nlst()` and `ftplib.FTP.dir()`: use `ftplib.FTP.mlsd()`
- `platform.popen()`: use the `subprocess` module. Check especially the *Replacing Older Functions with the subprocess Module* section (issue 11377).
- issue 13374: The Windows bytes API has been deprecated in the `os` module. Use Unicode filenames, instead of bytes filenames, to not depend on the ANSI code page anymore and to support any filename.
- issue 13988: The `xml.etree.cElementTree` module is deprecated. The accelerator is used automatically whenever available.
- The behaviour of `time.clock()` depends on the platform: use the new `time.perf_counter()` or `time.process_time()` function instead, depending on your requirements, to have a well defined behaviour.
- The `os.stat_float_times()` function is deprecated.
- `abc` module:
  - `abc.abstractproperty` has been deprecated, use `property` with `abc.abstractmethod()` instead.
  - `abc.abstractclassmethod` has been deprecated, use `classmethod` with `abc.abstractmethod()` instead.
  - `abc.abstractstaticmethod` has been deprecated, use `staticmethod` with `abc.abstractmethod()` instead.
- `importlib` package:
  - `importlib.abc.SourceLoader.path_mtime()` is now deprecated in favour of `importlib.abc.SourceLoader.path_stats()` as bytecode files now store both the modification time and size of the source file the bytecode file was compiled from.

Deprecated functions and types of the C API

The `Py_UNICODE` has been deprecated by **PEP 393** and will be removed in Python 4.

All functions using this type are deprecated:

Unicode functions and methods using `Py_UNICODE` and `Py_UNICODE*` types:

- `PyUnicode_FromUnicode`: use `PyUnicode_FromWideChar()` or `PyUnicode_FromKindAndData()`
- `PyUnicode_AS_UNICODE`, `PyUnicode_AsUnicode()`, `PyUnicode_AsUnicodeAndSize()`: use `PyUnicode_AsWideCharString()`
- `PyUnicode_AS_DATA`: use `PyUnicode_DATA` with `PyUnicode_READ` and `PyUnicode_WRITE`
- `PyUnicode_GET_SIZE`, `PyUnicode_GetSize()`: use `PyUnicode_GET_LENGTH` or `PyUnicode_GetLength()`
- `PyUnicode_GET_DATA_SIZE`: use `PyUnicode_GET_LENGTH(str) * PyUnicode_KIND(str)` (only work on ready strings)
- `PyUnicode_AsUnicodeCopy()`: use `PyUnicode_AsUCS4Copy()` or `PyUnicode_AsWideCharString()`
- `PyUnicode_GetMax()`

Functions and macros manipulating Py_UNICODE* strings:

- `Py_UNICODE_strlen`: use `PyUnicode_GetLength()` or `PyUnicode_GET_LENGTH`
- `Py_UNICODE_strcat`: use `PyUnicode_CopyCharacters()` or `PyUnicode_FromFormat()`
- `Py_UNICODE_strcpy`, `Py_UNICODE_strncpy`, `Py_UNICODE_COPY`: use `PyUnicode_CopyCharacters()` or `PyUnicode_Substring()`
- `Py_UNICODE_strcmp`: use `PyUnicode_Compare()`
- `Py_UNICODE_strncmp`: use `PyUnicode_Tailmatch()`
- `Py_UNICODE_strchr`, `Py_UNICODE_strrchr`: use `PyUnicode_FindChar()`
- `Py_UNICODE_FILL`: use `PyUnicode_Fill()`
- `Py_UNICODE_MATCH`

Encoders:

- `PyUnicode_Encode()`: use `PyUnicode_AsEncodedObject()`
- `PyUnicode_EncodeUTF7()`
- `PyUnicode_EncodeUTF8()`: use `PyUnicode_AsUTF8()` or `PyUnicode_AsUTF8String()`
- `PyUnicode_EncodeUTF32()`
- `PyUnicode_EncodeUTF16()`
- `PyUnicode_EncodeUnicodeEscape:()` use `PyUnicode_AsUnicodeEscapeString()`

- PyUnicode_EncodeRawUnicodeEscape:() use PyUnicode_AsRawUnicodeEscapeString()
- PyUnicode_EncodeLatin1(): use PyUnicode_AsLatin1String()
- PyUnicode_EncodeASCII(): use PyUnicode_AsASCIIString()
- PyUnicode_EncodeCharmap()
- PyUnicode_TranslateCharmap()
- PyUnicode_EncodeMBCS(): use PyUnicode_AsMBCSString() or PyUnicode_EncodeCodePage() (with CP_ACP code_page)
- PyUnicode_EncodeDecimal(), PyUnicode_TransformDecimalToASCII()

Deprecated features

The array module's 'u' format code is now deprecated and will be removed in Python 4 together with the rest of the (Py_UNICODE) API.

Porting to Python 3.3

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

Porting Python code

- Hash randomization is enabled by default. Set the PYTHONHASHSEED environment variable to 0 to disable hash randomization. See also the object.__hash__() method.
- issue 12326: On Linux, sys.platform doesn't contain the major version anymore. It is now always 'linux', instead of 'linux2' or 'linux3' depending on the Linux version used to build Python. Replace sys.platform == 'linux2' with sys.platform.startswith('linux'), or directly sys.platform == 'linux' if you don't need to support older Python versions.
- issue 13847, issue 14180: time and datetime: OverflowError is now raised instead of ValueError if a timestamp is out of range. OSError is now raised if C functions gmtime() or localtime() failed.
- The default finders used by import now utilize a cache of what is contained within a specific directory. If you create a Python source file or sourceless bytecode file, make sure to call importlib.invalidate_caches() to clear out the cache for the finders to notice the new file.
- ImportError now uses the full name of the module that was attemped to be

imported. Doctests that check ImportErrors' message will need to be updated to use the full name of the module instead of just the tail of the name.

- The *index* argument to `__import__()` now defaults to 0 instead of -1 and no longer support negative values. It was an oversight when **PEP 328** was implemented that the default value remained -1. If you need to continue to perform a relative import followed by an absolute import, then perform the relative import using an index of 1, followed by another import using an index of 0. It is preferred, though, that you use `importlib.import_module()` rather than call `__import__()` directly.
- `__import__()` no longer allows one to use an index value other than 0 for top-level modules. E.g. `__import__('sys', level=1)` is now an error.
- Because `sys.meta_path` and `sys.path_hooks` now have finders on them by default, you will most likely want to use `list.insert()` instead of `list.append()` to add to those lists.
- Because `None` is now inserted into `sys.path_importer_cache`, if you are clearing out entries in the dictionary of paths that do not have a finder, you will need to remove keys paired with values of `None` **and** `imp.NullImporter` to be backwards-compatible. This will lead to extra overhead on older versions of Python that re-insert `None` into `sys.path_importer_cache` where it repesents the use of implicit finders, but semantically it should not change anything.
- `importlib.abc.Finder` no longer specifies a *find_module()* abstract method that must be implemented. If you were relying on subclasses to implement that method, make sure to check for the method's existence first. You will probably want to check for *find_loader()* first, though, in the case of working with *path entry finders*.
- `pkgutil` has been converted to use `importlib` internally. This eliminates many edge cases where the old behaviour of the PEP 302 import emulation failed to match the behaviour of the real import system. The import emulation itself is still present, but is now deprecated. The `pkgutil.iter_importers()` and `pkgutil.walk_packages()` functions special case the standard import hooks so they are still supported even though they do not provide the non-standard `iter_modules()` method.
- A longstanding RFC-compliance bug (issue 1079) in the parsing done by `email.header.decode_header()` has been fixed. Code that uses the standard idiom to convert encoded headers into unicode (`str(make_header(decode_header(h))`) will see no change, but code that looks at the individual tuples returned by decode_header will see that whitespace that precedes or follows `ASCII` sections is now included in the `ASCII` section. Code that builds headers using `make_header` should also continue to work without change, since `make_header` continues to add whitespace between `ASCII` and

non-`ASCII` sections if it is not already present in the input strings.

- `email.utils.formataddr()` now does the correct content transfer encoding when passed non-`ASCII` display names. Any code that depended on the previous buggy behavior that preserved the non-`ASCII` unicode in the formatted output string will need to be changed (issue 1690608).
- `poplib.POP3.quit()` may now raise protocol errors like all other `poplib` methods. Code that assumes `quit` does not raise `poplib.error_proto` errors may need to be changed if errors on `quit` are encountered by a particular application (issue 11291).
- The `strict` argument to `email.parser.Parser`, deprecated since Python 2.4, has finally been removed.
- The deprecated method `unittest.TestCase.assertSameElements` has been removed.
- The deprecated variable `time.accept2dyear` has been removed.
- The deprecated `Context._clamp` attribute has been removed from the `decimal` module. It was previously replaced by the public attribute `clamp`. (See issue 8540.)
- The undocumented internal helper class `SSLFakeFile` has been removed from `smtplib`, since its functionality has long been provided directly by `socket.socket.makefile()`.
- Passing a negative value to `time.sleep()` on Windows now raises an error instead of sleeping forever. It has always raised an error on posix.
- The `ast.__version__` constant has been removed. If you need to make decisions affected by the AST version, use `sys.version_info` to make the decision.
- Code that used to work around the fact that the `threading` module used factory functions by subclassing the private classes will need to change to subclass the now-public classes.
- The undocumented debugging machinery in the threading module has been removed, simplifying the code. This should have no effect on production code, but is mentioned here in case any application debug frameworks were interacting with it (issue 13550).

Porting C code

- In the course of changes to the buffer API the undocumented `smalltable` member of the `Py_buffer` structure has been removed and the layout of the `PyMemoryViewObject` has changed.

All extensions relying on the relevant parts in `memoryobject.h` or `object.h` must be rebuilt.

- Due to *PEP 393*, the `Py_UNICODE` type and all functions using this type are deprecated (but will stay available for at least five years). If you were using low-level Unicode APIs to construct and access unicode objects and you want to benefit of the memory footprint reduction provided by PEP 393, you have to convert your code to the new *Unicode API*.

  However, if you only have been using high-level functions such as `PyUnicode_Concat()`, `PyUnicode_Join()` or `PyUnicode_FromFormat()`, your code will automatically take advantage of the new unicode representations.

- `PyImport_GetMagicNumber()` now returns -1 upon failure.

- As a negative value for the *level* argument to `__import__()` is no longer valid, the same now holds for `PyImport_ImportModuleLevel()`. This also means that the value of *level* used by `PyImport_ImportModuleEx()` is now 0 instead of -1.

Building C extensions

- The range of possible file names for C extensions has been narrowed. Very rarely used spellings have been suppressed: under POSIX, files named `xxxmodule.so`, `xxxmodule.abi3.so` and `xxxmodule.cpython-*.so` are no longer recognized as implementing the `xxx` module. If you had been generating such files, you have to switch to the other spellings (i.e., remove the `module` string from the file names).

  (implemented in issue 14040.)

Command Line Switch Changes

- The -Q command-line flag and related artifacts have been removed. Code checking sys.flags.division_warning will need updating.

  (issue 10998, contributed by Éric Araujo.)

- When **python** is started with *-S*, `import site` will no longer add site-specific paths to the module search paths. In previous versions, it did.

(issue 11591, contributed by Carl Meyer with editions by Éric Araujo.)

What's New In Python 3.4

Author: R. David Murray <rdmurray@bitdance.com> (Editor)

This article explains the new features in Python 3.4, compared to 3.3. Python 3.4 was released on March 16, 2014. For full details, see the changelog.

See also: **PEP 429** – Python 3.4 Release Schedule

Summary – Release Highlights

New syntax features:

- No new syntax features were added in Python 3.4.

Other new features:

- *pip should always be available* (**PEP 453**).
- *Newly created file descriptors are non-inheritable* (**PEP 446**).
- command line option for *isolated mode* (issue 16499).
- *improvements in the handling of codecs* that are not text encodings (multiple issues).
- *A ModuleSpec Type* for the Import System (**PEP 451**). (Affects importer authors.)
- The `marshal` format has been made *more compact and efficient* (issue 16475).

New library modules:

- `asyncio`: *New provisional API for asynchronous IO* (**PEP 3156**).
- `ensurepip`: *Bootstrapping the pip installer* (**PEP 453**).
- `enum`: *Support for enumeration types* (**PEP 435**).
- `pathlib`: *Object-oriented filesystem paths* (**PEP 428**).
- `selectors`: *High-level and efficient I/O multiplexing*, built upon the `select` module primitives (part of **PEP 3156**).
- `statistics`: A basic *numerically stable statistics library* (**PEP 450**).
- `tracemalloc`: *Trace Python memory allocations* (**PEP 454**).

Significantly improved library modules:

- *Single-dispatch generic functions* in `functools` (**PEP 443**).

- New `pickle` *protocol 4* (**PEP 3154**).
- `multiprocessing` now has *an option to avoid using os.fork on Unix* (issue 8713).
- `email` has a new submodule, `contentmanager`, and a new `Message` subclass (`EmailMessage`) that *simplify MIME handling* (issue 18891).
- The `inspect` and `pydoc` modules are now capable of correct introspection of a much wider variety of callable objects, which improves the output of the Python `help()` system.
- The `ipaddress` module API has been declared stable

Security improvements:

- *Secure and interchangeable hash algorithm* (**PEP 456**).
- *Make newly created file descriptors non-inheritable* (**PEP 446**) to avoid leaking file descriptors to child processes.
- New command line option for *isolated mode*, (issue 16499).
- `multiprocessing` now has *an option to avoid using os.fork on Unix*. *spawn* and *forkserver* are more secure because they avoid sharing data with child processes.
- `multiprocessing` child processes on Windows no longer inherit all of the parent's inheritable handles, only the necessary ones.
- A new `hashlib.pbkdf2_hmac()` function provides the PKCS#5 password-based key derivation function 2.
- *TLSv1.1 and TLSv1.2 support* for `ssl`.
- *Retrieving certificates from the Windows system cert store support* for `ssl`.
- *Server-side SNI (Server Name Indication) support* for `ssl`.
- The `ssl.SSLContext` class has a *lot of improvements*.
- All modules in the standard library that support SSL now support server certificate verification, including hostname matching (`ssl.match_hostname()`) and CRLs (Certificate Revocation lists, see `ssl.SSLContext.load_verify_locations()`).

CPython implementation improvements:

- *Safe object finalization* (**PEP 442**).
- Leveraging **PEP 442**, in most cases *module globals are no longer set to None during finalization* (issue 18214).
- *Configurable memory allocators* (**PEP 445**).
- *Argument Clinic* (**PEP 436**).

Please read on for a comprehensive list of user-facing changes, including many other smaller improvements, CPython optimizations, deprecations, and potential porting issues.

New Features

PEP 453: Explicit Bootstrapping of PIP in Python Installations

Bootstrapping pip By Default

The new `ensurepip` module (defined in **PEP 453**) provides a standard cross-platform mechanism to bootstrap the pip installer into Python installations and virtual environments. The version of `pip` included with Python 3.4.0 is `pip` 1.5.4, and future 3.4.x maintenance releases will update the bundled version to the latest version of `pip` that is available at the time of creating the release candidate.

By default, the commands `pipX` and `pipX.Y` will be installed on all platforms (where X.Y stands for the version of the Python installation), along with the `pip` Python package and its dependencies. On Windows and in virtual environments on all platforms, the unversioned `pip` command will also be installed. On other platforms, the system wide unversioned `pip` command typically refers to the separately installed Python 2 version.

The *pyvenv* command line utility and the `venv` module make use of the `ensurepip` module to make `pip` readily available in virtual environments. When using the command line utility, `pip` is installed by default, while when using the `venv` module *API* installation of `pip` must be requested explicitly.

For CPython *source builds on POSIX systems*, the `make install` and `make altinstall` commands bootstrap `pip` by default. This behaviour can be controlled through configure options, and overridden through Makefile options.

On Windows and Mac OS X, the CPython installers now default to installing `pip` along with CPython itself (users may opt out of installing it during the installation process). Window users will need to opt in to the automatic `PATH` modifications to have `pip` available from the command line by default, otherwise it can still be accessed through the Python launcher for Windows as `py -m pip`.

As discussed in the PEP, platform packagers may choose not to install these commands by default, as long as, when invoked, they provide clear and simple directions on how to install them on that platform (usually using the system package manager).

Note: To avoid conflicts between parallel Python 2 and Python 3 installations, only the versioned `pip3` and `pip3.4` commands are bootstrapped by default when `ensurepip` is invoked directly – the `--default-pip` option is needed to also request the unversioned `pip` command. `pyvenv` and the Windows installer ensure that the unqualified `pip` command is made available in those environments, and `pip` can always be invoked via the `-m` switch rather than directly to avoid ambiguity on systems with multiple Python installations.

Documentation Changes

As part of this change, the *Installing Python Modules* and *Distributing Python Modules* sections of the documentation have been completely redesigned as short getting started and FAQ documents. Most packaging documentation has now been moved out to the Python Packaging Authority maintained Python Packaging User Guide and the documentation of the individual projects.

However, as this migration is currently still incomplete, the legacy versions of those guides remaining available as *Installing Python Modules (Legacy version)* and *Distributing Python Modules (Legacy version)*.

See also:

PEP 453 – Explicit bootstrapping of pip in Python installations
PEP written by Donald Stufft and Nick Coghlan, implemented by Donald Stufft, Nick Coghlan, Martin von Löwis and Ned Deily.

PEP 446: Newly Created File Descriptors Are Non-Inheritable

PEP 446 makes newly created file descriptors *non-inheritable*. In general, this is the behavior an application will want: when launching a new process, having currently open files also open in the new process can lead to all sorts of hard to find bugs, and potentially to security issues.

However, there are occasions when inheritance is desired. To support these cases, the following new functions and methods are available:

- `os.get_inheritable()`, `os.set_inheritable()`
- `os.get_handle_inheritable()`, `os.set_handle_inheritable()`
- `socket.socket.get_inheritable()`, `socket.socket.set_inheritable()`

See also:

PEP 446 - Make newly created file descriptors non-inheritable
PEP written and implemented by Victor Stinner.

Improvements to Codec Handling

Since it was first introduced, the `codecs` module has always been intended to operate as a type-neutral dynamic encoding and decoding system. However, its close coupling with the Python text model, especially the type restricted convenience methods on the builtin `str`, `bytes` and `bytearray` types, has historically obscured that fact.

As a key step in clarifying the situation, the `codecs.encode()` and `codecs.decode()` convenience functions are now properly documented in Python 2.7, 3.3 and 3.4. These functions have existed in the `codecs` module (and have been covered by the regression test suite) since Python 2.4, but were previously only discoverable through runtime introspection.

Unlike the convenience methods on `str`, `bytes` and `bytearray`, the `codecs` convenience functions support arbitrary codecs in both Python 2 and Python 3, rather than being limited to Unicode text encodings (in Python 3) or `basestring` <-> `basestring` conversions (in Python 2).

In Python 3.4, the interpreter is able to identify the known non-text encodings provided in the standard library and direct users towards these general purpose convenience functions when appropriate:

```
>>> b"abcdef".decode("hex")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LookupError: 'hex' is not a text encoding; use codecs.decode() to handle a

>>> "hello".encode("rot13")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
```

```
LookupError: 'rot13' is not a text encoding; use codecs.encode() to handle

>>> open("foo.txt", encoding="hex")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
LookupError: 'hex' is not a text encoding; use codecs.open() to handle ark
```

In a related change, whenever it is feasible without breaking backwards compatibility, exceptions raised during encoding and decoding operations are wrapped in a chained exception of the same type that mentions the name of the codec responsible for producing the error:

```
>>> import codecs

>>> codecs.decode(b"abcdefgh", "hex")
Traceback (most recent call last):
  File "/usr/lib/python3.4/encodings/hex_codec.py", line 20, in hex_decode
    return (binascii.a2b_hex(input), len(input))
binascii.Error: Non-hexadecimal digit found

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
binascii.Error: decoding with 'hex' codec failed (Error: Non-hexadecimal c

>>> codecs.encode("hello", "bz2")
Traceback (most recent call last):
  File "/usr/lib/python3.4/encodings/bz2_codec.py", line 17, in bz2_encode
    return (bz2.compress(input), len(input))
  File "/usr/lib/python3.4/bz2.py", line 498, in compress
    return comp.compress(data) + comp.flush()
TypeError: 'str' does not support the buffer interface

The above exception was the direct cause of the following exception:

Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: encoding with 'bz2' codec failed (TypeError: 'str' does not sup
```

Finally, as the examples above show, these improvements have permitted the restoration of the convenience aliases for the non-Unicode codecs that were themselves restored in Python 3.2. This means that encoding binary data to and from its hexadecimal representation (for example) can now be written as:

```
>>> from codecs import encode, decode
>>> encode(b"hello", "hex")
b'68656c6c6f'
>>> decode(b"68656c6c6f", "hex")
b'hello'
```

The binary and text transforms provided in the standard library are detailed in *Binary Transforms* and *Text Transforms*.

(Contributed by Nick Coghlan in issue 7475, issue 17827, issue 17828 and issue 19619)

PEP 451: A ModuleSpec Type for the Import System

PEP 451 provides an encapsulation of the information about a module that the import machinery will use to load it (that is, a module specification). This helps simplify both the import implementation and several import-related APIs. The change is also a stepping stone for several future import-related improvements.

The public-facing changes from the PEP are entirely backward-compatible. Furthermore, they should be transparent to everyone but importer authors. Key finder and loader methods have been deprecated, but they will continue working. New importers should use the new methods described in the PEP. Existing importers should be updated to implement the new methods. See the *Deprecated* section for a list of methods that should be replaced and their replacements.

Other Language Changes

Some smaller changes made to the core Python language are:

- Unicode database updated to UCD version 6.3.
- `min()` and `max()` now accept a *default* keyword-only argument that can be used to specify the value they return if the iterable they are evaluating has no elements. (Contributed by Julian Berman in issue 18111.)
- Module objects are now `weakref`'able.
- Module `__file__` attributes (and related values) should now always contain absolute paths by default, with the sole exception of `__main__.__file__` when a script has been executed directly using a relative path (Contributed by Brett Cannon in issue 18416).
- All the UTF-* codecs (except UTF-7) now reject surrogates during both

encoding and decoding unless the `surrogatepass` error handler is used, with the exception of the UTF-16 decoder (which accepts valid surrogate pairs) and the UTF-16 encoder (which produces them while encoding non-BMP characters). Contributed by Victor Stinner, Kang-Hao (Kenny) Lu and Serhiy Storchaka in issue 12892.

- New German EBCDIC *codec* `cp273`. (Contributed by Michael Bierenfeld and Andrew Kuchling in issue 1097797.)
- New Ukrainian *codec* `cp1125`. (Contributed by Serhiy Storchaka in issue 19668.)
- `bytes`.join() and `bytearray`.join() now accept arbitrary buffer objects as arguments. (Contributed by Antoine Pitrou in issue 15958.)
- The `int` constructor now accepts any object that has an `__index__` method for its *base* argument. (Contributed by Mark Dickinson in issue 16772.)
- Frame objects now have a `clear()` method that clears all references to local variables from the frame. (Contributed by Antoine Pitrou in issue 17934.)
- `memoryview` is now registered as a `Sequence`, and supports the `reversed()` builtin. (Contributed by Nick Coghlan and Claudiu Popa in issue 18690 and issue 19078.)
- Signatures reported by `help()` have been modified and improved in several cases as a result of the introduction of Argument Clinic and other changes to the `inspect` and `pydoc` modules.
- `__length_hint__()` is now part of the formal language specification (see **PEP 424**). (Contributed by Armin Ronacher in issue 16148.)

New Modules

asyncio

The new `asyncio` module (defined in **PEP 3156**) provides a standard pluggable event loop model for Python, providing solid asynchronous IO support in the standard library, and making it easier for other event loop implementations to interoperate with the standard library and each other.

For Python 3.4, this module is considered a *provisional API*.

See also:

PEP 3156 – Asynchronous IO Support Rebooted: the "asyncio" Module
PEP written and implementation led by Guido van Rossum.

ensurepip

The new `ensurepip` module is the primary infrastructure for the **PEP 453** implementation. In the normal course of events end users will not need to interact with this module, but it can be used to manually bootstrap `pip` if the automated bootstrapping into an installation or virtual environment was declined.

`ensurepip` includes a bundled copy of `pip`, up-to-date as of the first release candidate of the release of CPython with which it ships (this applies to both maintenance releases and feature releases). `ensurepip` does not access the internet. If the installation has Internet access, after `ensurepip` is run the bundled `pip` can be used to upgrade `pip` to a more recent release than the bundled one. (Note that such an upgraded version of `pip` is considered to be a separately installed package and will not be removed if Python is uninstalled.)

The module is named *ensure*pip because if called when `pip` is already installed, it does nothing. It also has an `--upgrade` option that will cause it to install the bundled copy of `pip` if the existing installed version of `pip` is older than the bundled copy.

enum

The new `enum` module (defined in **PEP 435**) provides a standard implementation of enumeration types, allowing other modules (such as `socket`) to provide more informative error messages and better debugging support by replacing opaque integer constants with backwards compatible enumeration values.

See also:

PEP 435 – Adding an Enum type to the Python standard library
PEP written by Barry Warsaw, Eli Bendersky and Ethan Furman, implemented by Ethan Furman.

pathlib

The new `pathlib` module offers classes representing filesystem paths with semantics appropriate for different operating systems. Path classes are divided between *pure paths*, which provide purely computational operations without I/O, and *concrete* *paths*, which inherit from pure paths but also provide I/O operations.

For Python 3.4, this module is considered a *provisional API*.

See also:

PEP 428 – The pathlib module – object-oriented filesystem paths
PEP written and implemented by Antoine Pitrou.

selectors

The new `selectors` module (created as part of implementing **PEP 3156**) allows high-level and efficient I/O multiplexing, built upon the `select` module primitives.

statistics

The new `statistics` module (defined in **PEP 450**) offers some core statistics functionality directly in the standard library. This module supports calculation of the mean, median, mode, variance and standard deviation of a data series.

See also:

PEP 450 – Adding A Statistics Module To The Standard Library
PEP written and implemented by Steven D'Aprano

tracemalloc

The new `tracemalloc` module (defined in **PEP 454**) is a debug tool to trace memory blocks allocated by Python. It provides the following information:

- Trace where an object was allocated
- Statistics on allocated memory blocks per filename and per line number: total size, number and average size of allocated memory blocks
- Compute the differences between two snapshots to detect memory leaks

See also:

PEP 454 – Add a new tracemalloc module to trace Python memory allocations

PEP written and implemented by Victor Stinner

Improved Modules

abc

New function `abc.get_cache_token()` can be used to know when to invalidate caches that are affected by changes in the object graph. (Contributed by Łukasz Langa in issue 16832.)

New class `ABC` has `ABCMeta` as its meta class. Using `ABC` as a base class has essentially the same effect as specifying `metaclass=abc.ABCMeta`, but is simpler to type and easier to read. (Contributed by Bruno Dupuis in issue 16049.)

aifc

The `getparams()` method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in issue 17818.)

`aifc.open()` now supports the context manager protocol: when used in a `with` block, the `close()` method of the returned object will be called automatically at the end of the block. (Contributed by Serhiy Storchacha in issue 16486.)

The `writeframesraw()` and `writeframes()` methods now accept any *bytes-like object*. (Contributed by Serhiy Storchaka in issue 8311.)

argparse

The `FileType` class now accepts *encoding* and *errors* arguments, which are passed through to `open()`. (Contributed by Lucas Maystre in issue 11175.)

audioop

`audioop` now supports 24-bit samples. (Contributed by Serhiy Storchaka in issue 12866.)

New `byteswap()` function converts big-endian samples to little-endian and vice versa

(Contributed by Serhiy Storchaka in issue 19641).

All `audioop` functions now accept any *bytes-like object*. Strings are not accepted: they didn't work before, now they raise an error right away. (Contributed by Serhiy Storchaka in issue 16685.)

base64

The encoding and decoding functions in `base64` now accept any *bytes-like object* in cases where it previously required a `bytes` or `bytearray` instance. (Contributed by Nick Coghlan in issue 17839.)

New functions `a85encode()`, `a85decode()`, `b85encode()`, and `b85decode()` provide the ability to encode and decode binary data from and to `Ascii85` and the git/mercurial `Base85` formats, respectively. The `a85` functions have options that can be used to make them compatible with the variants of the `Ascii85` encoding, including the Adobe variant. (Contributed by Martin Morrison, the Mercurial project, Serhiy Storchaka, and Antoine Pitrou in issue 17618.)

collections

The `ChainMap.new_child()` method now accepts an *m* argument specifying the child map to add to the chain. This allows an existing mapping and/or a custom mapping type to be used for the child. (Contributed by Vinay Sajip in issue 16613.)

colorsys

The number of digits in the coefficients for the RGB — YIQ conversions have been expanded so that they match the FCC NTSC versions. The change in results should be less than 1% and may better match results found elsewhere. (Contributed by Brian Landers and Serhiy Storchaka in issue 14323.)

contextlib

The new `contextlib.suppress` context manager helps to clarify the intent of code that deliberately suppresses exceptions from a single statement. (Contributed by Raymond Hettinger in issue 15806 and Zero Piraeus in issue 19266)

The new contextlib.redirect_stdout() context manager makes it easier for utility scripts to handle inflexible APIs that write their output to sys.stdout and don't provide any options to redirect it. Using the context manager, the sys.stdout output can be redirected to any other stream or, in conjunction with io.StringIO, to a string. The latter can be especially useful, for example, to capture output from a function that was written to implement a command line interface. It is recommended only for utility scripts because it affects the global state of sys.stdout. (Contributed by Raymond Hettinger in issue 15805)

The contextlib documentation has also been updated to include a *discussion* of the differences between single use, reusable and reentrant context managers.

dbm

dbm.open() objects now support the context management protocol. When used in a with statement, the close method of the database object will be called automatically at the end of the block. (Contributed by Claudiu Popa and Nick Coghlan in issue 19282.)

dis

Functions show_code(), dis(), distb(), and disassemble() now accept a keyword-only *file* argument that controls where they write their output.

The dis module is now built around an Instruction class that provides object oriented access to the details of each individual bytecode operation.

A new method, get_instructions(), provides an iterator that emits the Instruction stream for a given piece of Python code. Thus it is now possible to write a program that inspects and manipulates a bytecode object in ways different from those provided by the dis module itself. For example:

```
>>> import dis
>>> for instr in dis.get_instructions(lambda x: x + 1):
...     print(instr.opname)
LOAD_FAST
LOAD_CONST
BINARY_ADD
RETURN_VALUE
```

The various display tools in the `dis` module have been rewritten to use these new components.

In addition, a new application-friendly class `Bytecode` provides an object-oriented API for inspecting bytecode in both in human-readable form and for iterating over instructions. The `Bytecode` constructor takes the same arguments that `get_instruction()` does (plus an optional *current_offset*), and the resulting object can be iterated to produce `Instruction` objects. But it also has a `dis` method, equivalent to calling `dis` on the constructor argument, but returned as a multi-line string:

```
>>> bytecode = dis.Bytecode(lambda x: x +1, current_offset=3)
>>> for instr in bytecode:
...     print('{} ({})'.format(instr.opname, instr.opcode))
LOAD_FAST (124)
LOAD_CONST (100)
BINARY_ADD (23)
RETURN_VALUE (83)
>>> bytecode.dis().splitlines()
['  1           0 LOAD_FAST                0 (x)',
 '      -->     3 LOAD_CONST               1 (1)',
 '              6 BINARY_ADD',
 '              7 RETURN_VALUE']
```

`Bytecode` also has a class method, `from_traceback()`, that provides the ability to manipulate a traceback (that is, `print(Bytecode.from_traceback(tb).dis())` is equivalent to `distb(tb)`).

(Contributed by Nick Coghlan, Ryan Kelly and Thomas Kluyver in issue 11816 and Claudiu Popa in issue 17916)

New function `stack_effect()` computes the effect on the Python stack of a given opcode and argument, information that is not otherwise available. (Contributed by Larry Hastings in issue 19722.)

doctest

A new *option flag*, `FAIL_FAST`, halts test running as soon as the first failure is detected. (Contributed by R. David Murray and Daniel Urban in issue 16522.)

The `doctest` command line interface now uses `argparse`, and has two new options, `-o` and `-f`. `-o` allows *doctest options* to be specified on the command line, and `-f` is a shorthand for `-o FAIL_FAST` (to parallel the similar option supported by the `unittest` CLI). (Contributed by R. David Murray in issue 11390.)

`doctest` will now find doctests in extension module `__doc__` strings. (Contributed by Zachary Ware in issue 3158.)

email

`as_string()` now accepts a *policy* argument to override the default policy of the message when generating a string representation of it. This means that `as_string` can now be used in more circumstances, instead of having to create and use a `generator` in order to pass formatting parameters to its `flatten` method. (Contributed by R. David Murray in issue 18600.)

New method `as_bytes()` added to produce a bytes representation of the message in a fashion similar to how `as_string` produces a string representation. It does not accept the *maxheaderlen* argument, but does accept the *unixfrom* and *policy* arguments. The `Message __bytes__()` method calls it, meaning that `bytes(mymsg)` will now produce the intuitive result: a bytes object containing the fully formatted message. (Contributed by R. David Murray in issue 18600.)

The `Message.set_param()` message now accepts a *replace* keyword argument. When specified, the associated header will be updated without changing its location in the list of headers. For backward compatibility, the default is `False`. (Contributed by R. David Murray in issue 18891.)

A pair of new subclasses of `Message` have been added (`EmailMessage` and `MIMEPart`), along with a new sub-module, `contentmanager` and a new `policy` attribute `content_manager`. All documentation is currently in the new module, which is being added as part of email's new *provisional API*. These classes provide a number of new methods that make extracting content from and inserting content into email messages much easier. For details, see the `contentmanager` documentation and the *Examples using the Provisional API*. These API additions complete the bulk of the work that was planned as part of the email6 project. The currently provisional API is scheduled to become final in Python 3.5 (possibly with a few minor additions in the area of error handling). (Contributed by R. David Murray in issue 18891.)

filecmp

A new `clear_cache()` function provides the ability to clear the `filecmp` comparison cache, which uses `os.stat()` information to determine if the file has changed since the last compare. This can be used, for example, if the file might have been changed and re-checked in less time than the resolution of a particular filesystem's file modification time field. (Contributed by Mark Levitt in issue 18149.)

New module attribute `DEFAULT_IGNORES` provides the list of directories that are used as the default value for the *ignore* parameter of the `dircmp()` function. (Contributed by Eli Bendersky in issue 15442.)

functools

The new `partialmethod()` descriptor brings partial argument application to descriptors, just as `partial()` provides for normal callables. The new descriptor also makes it easier to get arbitrary callables (including `partial()` instances) to behave like normal instance methods when included in a class definition. (Contributed by Alon Horev and Nick Coghlan in issue 4331)

The new `singledispatch()` decorator brings support for single-dispatch generic functions to the Python standard library. Where object oriented programming focuses on grouping multiple operations on a common set of data into a class, a generic function focuses on grouping multiple implementations of an operation that allows it to work with *different* kinds of data.

See also:

PEP 443 – Single-dispatch generic functions
PEP written and implemented by Łukasz Langa.

`total_ordering()` now supports a return value of `NotImplemented` from the underlying comparison function. (Contributed by Katie Miller in issue 10042.)

A pure-python version of the `partial()` function is now in the stdlib; in CPython it is overridden by the C accelerated version, but it is available for other implementations to use. (Contributed by Brian Thorne in issue 12428.)

gc

New function `get_stats()` returns a list of three per-generation dictionaries containing the collections statistics since interpreter startup. (Contributed by Antoine Pitrou in issue 16351.)

glob

A new function `escape()` provides a way to escape special characters in a filename so that they do not become part of the globbing expansion but are instead matched literally. (Contributed by Serhiy Storchaka in issue 8402.)

hashlib

A new `hashlib.pbkdf2_hmac()` function provides the PKCS#5 password-based key derivation function 2. (Contributed by Christian Heimes in issue 18582)

The `name` attribute of `hashlib` hash objects is now a formally supported interface. It has always existed in CPython's `hashlib` (although it did not return lower case names for all supported hashes), but it was not a public interface and so some other Python implementations have not previously supported it. (Contributed by Jason R. Coombs in issue 18532.)

hmac

`hmac` now accepts `bytearray` as well as `bytes` for the *key* argument to the `new()` function, and the *msg* parameter to both the `new()` function and the `update()` method now accepts any type supported by the `hashlib` module. (Contributed by Jonas Borgström in issue 18240.)

The *digestmod* argument to the `hmac.new()` function may now be any hash digest name recognized by `hashlib`. In addition, the current behavior in which the value of *digestmod* defaults to `MD5` is deprecated: in a future version of Python there will be no default value. (Contributed by Christian Heimes in issue 17276.)

With the addition of `block_size` and `name` attributes (and the formal documentation of the `digest_size` attribute), the `hmac` module now conforms fully to the **PEP 247**

API. (Contributed by Christian Heimes in issue 18775.)

html

New function `unescape()` function converts HTML5 character references to the corresponding Unicode characters. (Contributed by Ezio Melotti in issue 2927)

`HTMLParser` accepts a new keyword argument *convert_charrefs* that, when `True`, automatically converts all character references. For backward-compatibility, its value defaults to `False`, but it will change to `True` in a future version of Python, so you are invited to set it explicitly and update your code to use this new feature. (Contributed by Ezio Melotti in issue 13633)

The *strict* argument of `HTMLParser` is now deprecated. (Contributed by Ezio Melotti in issue 15114)

http

`send_error()` now accepts an optional additional *explain* parameter which can be used to provide an extended error description, overriding the hardcoded default if there is one. This extended error description will be formatted using the `error_message_format` attribute and sent as the body of the error response. (Contributed by Karl Cow in issue 12921.)

The `http.server` *command line interface* now has a `-b/--bind` option that causes the server to listen on a specific address. (Contributed by Malte Swart in issue 17764.)

importlib

The `InspectLoader` ABC defines a new method, `source_to_code()` that accepts source data and a path and returns a code object. The default implementation is equivalent to `compile(data, path, 'exec', dont_inherit=True)`. (Contributed by Eric Snow and Brett Cannon in issue 15627.)

`InspectLoader` also now has a default implementation for the `get_code()` method. However, it will normally be desirable to override the default implementation for performance reasons. (Contributed by Brett Cannon in issue 18072.)

The `reload()` function has been moved from `imp` to `importlib` as part of the `imp` module deprecation. (Contributed by Berker Peksag in issue 18193.)

`importlib.util` now has a `MAGIC_NUMBER` attribute providing access to the bytecode version number. This replaces the `get_magic()` function in the deprecated `imp` module. (Contributed by Brett Cannon in issue 18192.)

New `importlib.util` functions `cache_from_source()` and `source_from_cache()` replace the same-named functions in the deprecated `imp` module. (Contributed by Brett Cannon in issue 18194.)

The `importlib` bootstrap `NamespaceLoader` now conforms to the `InspectLoader` ABC, which means that `runpy` and `python -m` can now be used with namespace packages. (Contributed by Brett Cannon in issue 18058.)

`importlib.util` has a new function `decode_source()` that decodes source from bytes using universal newline processing. This is useful for implementing `InspectLoader.get_source()` methods.

`importlib.machinery.ExtensionFileLoader` now has a `get_filename()` method. This was inadvertently omitted in the original implementation. (Contributed by Eric Snow in issue 19152.)

inspect

The `inspect` module now offers a basic *command line interface* to quickly display source code and other information for modules, classes and functions. (Contributed by Claudiu Popa and Nick Coghlan in issue 18626)

`unwrap()` makes it easy to unravel wrapper function chains created by `functools.wraps()` (and any other API that sets the `__wrapped__` attribute on a wrapper function). (Contributed by Daniel Urban, Aaron Iles and Nick Coghlan in issue 13266)

As part of the implementation of the new `enum` module, the `inspect` module now has substantially better support for custom `__dir__` methods and dynamic class attributes provided through metaclasses (Contributed by Ethan Furman in issue 18929 and issue 19030) `getfullargspec()` and `getargspec()` now use the `signature()` API. This allows them to support a much broader range of callables, including those with `__signature__` attributes, those with metadata provided by argument clinic, `functools.partial()` objects and more. Note that, unlike `signature()`, these functions still ignore `__wrapped__` attributes, and report the already bound first argument for bound methods, so it is still necessary to update your code to use `signature()` directly if those features are desired. (Contributed by Yury Selivanov in issue 17481)

`signature()` now supports duck types of CPython functions, which adds support for functions compiled with Cython. (Contributed by Stefan Behnel and Yury Selivanov in issue 17159)

ipaddress

`ipaddress` was added to the standard library in Python 3.3 as a *provisional API*. With the release of Python 3.4, this qualification has been removed: `ipaddress` is now considered a stable API, covered by the normal standard library requirements to maintain backwards compatibility.

A new `is_global` property is `True` if an address is globally routeable. (Contributed by Peter Moody in issue 17400.)

logging

The `TimedRotatingFileHandler` has a new *atTime* parameter that can be used to specify the time of day when rollover should happen. (Contributed by Ronald Oussoren in issue 9556.)

`SocketHandler` and `DatagramHandler` now support Unix domain sockets (by setting *port* to `None`). (Contributed by Vinay Sajip in commit ce46195b56a9.)

`fileConfig()` now accepts a `configparser.RawConfigParser` subclass instance for the *fname* parameter. This facilitates using a configuration file when logging configuration is just a part of the overall application configuration, or where the application modifies the configuration before passing it to `fileConfig()`. (Contributed by Vinay Sajip in issue 16110.)

Logging configuration data received from a socket via the `logging.config.listen()` function can now be validated before being processed by supplying a verification function as the argument to the new *verify* keyword argument. (Contributed by Vinay Sajip in issue 15452.)

marshal

The default `marshal` version has been bumped to 3. The code implementing the new version restores the Python2 behavior of recording only one copy of interned strings and preserving the interning on deserialization, and extends this "one copy" ability to any object type (including handling recursive references). This reduces both the size of `.pyc` files and the amount of memory a module occupies in memory when it is loaded from a `.pyc` (or `.pyo`) file. (Contributed by Kristján Valur Jónsson in issue 16475, with additional speedups by Antoine Pitrou in issue 19219.)

mmap

mmap objects can now be `weakref`ed. (Contributed by Valerie Lambert in issue 4885.)

multiprocessing

On Unix two new *start methods*, (`spawn` and `forkserver`, have been added for starting processes using `multiprocessing`. These make the mixing of processes with threads more robust, and the `spawn` method matches the semantics that multiprocessing has always used on Windows. New function `get_all_start_methods()` reports all start methods available on the platform, `get_start_method()` reports the current start method, and `set_start_method()` sets the start method. (Contributed by Richard Oudkerk in issue 8713).

`multiprocessing` also now has the concept of a `context`, which determines how child processes are created. New function `get_context()` returns a context that uses a specified start method. It has the same API as the `multiprocessing` module itself, so you can use it to create `Pool`s and other objects that will operate within that context. This allows a framework and an application or different parts of the same application to use multiprocessing without interfering with each other. (Contributed by Richard Oudkerk in issue 18999.)

Except when using the old *fork* start method, child processes no longer inherit unneeded handles/file descriptors from their parents (part of issue 8713).

`multiprocessing` now relies on `runpy` (which implements the `-m` switch) to initialise `__main__` appropriately in child processes when using the `spawn` or `forkserver` start methods. This resolves some edge cases where combining multiprocessing, the `-m` command line switch, and explicit relative imports could cause obscure failures in child processes. (Contributed by Nick Coghlan in issue 19946)

operator

New function `length_hint()` provides an implementation of the specification for how the `__length_hint__()` special method should be used, as part of the **PEP 424** formal specification of this language feature. (Contributed by Armin Ronacher in issue 16148.)

There is now a pure-python version of the `operator` module available for reference and for use by alternate implementations of Python. (Contributed by Zachary Ware in issue 16694.)

os

There are new functions to get and set the *inheritable flag* of a file descriptor (`os.get_inheritable()`, `os.set_inheritable()`) or a Windows handle (`os.get_handle_inheritable()`, `os.set_handle_inheritable()`).

New function `cpu_count()` reports the number of CPUs available on the platform on which Python is running (or `None` if the count can’t be determined). The `multiprocessing.cpu_count()` function is now implemented in terms of this function). (Contributed by Trent Nelson, Yogesh Chaudhari, Victor Stinner, and Charles-François Natali in issue 17914.)

`os.path.samestat()` is now available on the Windows platform (and the `os.path.samefile()` implementation is now shared between Unix and Windows). (Contributed by Brian Curtin in issue 11939.)

`os.path.ismount()` now recognizes volumes mounted below a drive root on Windows. (Contributed by Tim Golden in issue 9035.)

`os.open()` supports two new flags on platforms that provide them, `O_PATH` (unopened file descriptor), and `O_TMPFILE` (unnamed temporary file; as of 3.4.0 release available only on Linux systems with a kernel version of 3.11 or newer that have uapi headers). (Contributed by Christian Heimes in issue 18673 and Benjamin Peterson, respectively.)

pdb

`pdb` has been enhanced to handle generators, `yield`, and `yield from` in a more useful fashion. This is especially helpful when debugging `asyncio` based programs. (Contributed by Andrew Svetlov and Xavier de Gaye in issue 16596.)

The `print` command has been removed from `pdb`, restoring access to the Python `print()` function from the pdb command line. Python2's `pdb` did not have a `print` command; instead, entering `print` executed the `print` statement. In Python3 `print` was mistakenly made an alias for the pdb `p` command. `p`, however, prints the `repr` of its argument, not the `str` like the Python2 `print` command did. Worse, the Python3 `pdb print` command shadowed the Python3 `print` function, making it inaccessible at the `pdb` prompt. (Contributed by Connor Osborn in issue 18764.)

pickle

`pickle` now supports (but does not use by default) a new pickle protocol, protocol 4. This new protocol addresses a number of issues that were present in previous protocols, such as the serialization of nested classes, very large strings and containers, and classes whose `__new__()` method takes keyword-only arguments. It also provides some efficiency improvements.

See also:

PEP 3154 – Pickle protocol 4
PEP written by Antoine Pitrou and implemented by Alexandre Vassalotti.

plistlib

`plistlib` now has an API that is similar to the standard pattern for stdlib serialization protocols, with new `load()`, `dump()`, `loads()`, and `dumps()` functions. (The older API is now deprecated.) In addition to the already supported XML plist format (`FMT_XML`), it also now supports the binary plist format (`FMT_BINARY`). (Contributed by Ronald Oussoren and others in issue 14455).

poplib

Two new methods have been added to `poplib`: `capa()`, which returns the list of capabilities advertised by the POP server, and `stls()`, which switches a clear-text POP3 session into an encrypted POP3 session if the POP server supports it. (Contributed by Lorenzo Catucci in issue 4473.)

pprint

The `pprint` module's `PrettyPrinter` class and its `pformat()`, and `pprint()` functions have a new option, *compact*, that controls how the output is formatted. Currently setting *compact* to `True` means that sequences will be printed with as many sequence elements as will fit within *width* on each (indented) line. (Contributed by Serhiy Storchaka in issue 19132.)

Long strings are now wrapped using Python's normal line continuation syntax. (Contributed by Antoine Pitrou in issue 17150).

pty

`pty.spawn()` now returns the status value from `os.waitpid()` on the child process, instead of `None`. (Contributed by Gregory P. Smith.)

pydoc

The `pydoc` module is now based directly on the `inspect.signature()` introspection API, allowing it to provide signature information for a wider variety of callable objects. This change also means that `__wrapped__` attributes are now taken into account when displaying help information (Contributed by Larry Hastings in issue 19674)

The `pydoc` module no longer displays the `self` parameter for already bound methods. Instead, it aims to always display the exact current signature of the supplied callable (Contributed by Larry Hastings in issue 20710)

In addition to the changes that have been made to `pydoc` directly, its handling of custom `__dir__` methods and various descriptor behaviours has also been improved substantially by the underlying changes in the `inspect` module.

As the `help()` builtin is based on `pydoc`, the above changes also affect the behaviour of `help()`.

re

New `fullmatch()` function and `regex.fullmatch()` method anchor the pattern at both ends of the string to match. This provides a way to be explicit about the goal of the match, which avoids a class of subtle bugs where `$` characters get lost during code changes or the addition of alternatives to an existing regular expression. (Contributed by Matthew Barnett in issue 16203.)

The repr of *regex objects* now includes the pattern and the flags; the repr of *match objects* now includes the start, end, and the part of the string that matched. (Contributed by Hugo Lopes Tavares and Serhiy Storchaka in issue 13592 and issue 17087.)

resource

New `prlimit()` function, available on Linux platforms with a kernel version of 2.6.36 or later and glibc of 2.13 or later, provides the ability to query or set the resource limits for processes other than the one making the call. (Contributed by Christian Heimes in issue 16595.)

On Linux kernel version 2.6.36 or later, there are there are also some new Linux specific constants: `RLIMIT_MSGQUEUE`, `RLIMIT_NICE`, `RLIMIT_RTPRIO`, `RLIMIT_RTTIME`, and `RLIMIT_SIGPENDING`. (Contributed by Christian Heimes in issue 19324.)

On FreeBSD version 9 and later, there some new FreeBSD specific constants: `RLIMIT_SBSIZE`, `RLIMIT_SWAP`, and `RLIMIT_NPTS`. (Contributed by Claudiu Popa in issue 19343.)

select

`epoll` objects now support the context management protocol. When used in a `with` statement, the `close()` method will be called automatically at the end of the block. (Contributed by Serhiy Storchaka in issue 16488.)

`devpoll` objects now have `fileno()` and `close()` methods, as well as a new attribute `closed`. (Contributed by Victor Stinner in issue 18794.)

shelve

`Shelf` instances may now be used in `with` statements, and will be automatically closed at the end of the `with` block. (Contributed by Filip Gruszczyński in issue 13896.)

shutil

`copyfile()` now raises a specific `Error` subclass, `SameFileError`, when the source and destination are the same file, which allows an application to take appropriate action on this specific error. (Contributed by Atsuo Ishimoto and Hynek Schlawack in issue 1492704.)

smtpd

The `SMTPServer` and `SMTPChannel` classes now accept a *map* keyword argument which, if specified, is passed in to `asynchat.async_chat` as its *map* argument. This allows an application to avoid affecting the global socket map. (Contributed by Vinay Sajip in issue 11959.)

smtplib

`SMTPException` is now a subclass of `OSError`, which allows both socket level errors and SMTP protocol level errors to be caught in one try/except statement by code that only cares whether or not an error occurred. (Contributed by Ned Jackson Lovely in issue 2118).

socket

The socket module now supports the `CAN_BCM` protocol on platforms that support it.

(Contributed by Brian Thorne in issue 15359.)

Socket objects have new methods to get or set their *inheritable flag*, `get_inheritable()` and `set_inheritable()`.

The `socket.AF_*` and `socket.SOCK_*` constants are now enumeration values using the new `enum` module. This allows meaningful names to be printed during debugging, instead of integer "magic numbers".

The `AF_LINK` constant is now available on BSD and OSX.

`inet_pton()` and `inet_ntop()` are now supported on Windows. (Contributed by Atsuo Ishimoto in issue 7171.)

sqlite3

A new boolean parameter to the `connect()` function, *uri*, can be used to indicate that the *database* parameter is a `uri` (see the SQLite URI documentation). (Contributed by poq in issue 13773.)

ssl

`PROTOCOL_TLSv1_1` and `PROTOCOL_TLSv1_2` (TLSv1.1 and TLSv1.2 support) have been added; support for these protocols is only available if Python is linked with OpenSSL 1.0.1 or later. (Contributed by Michele Orrù and Antoine Pitrou in issue 16692)

New function `create_default_context()` provides a standard way to obtain an `SSLContext` whose settings are intended to be a reasonable balance between compatibility and security. These settings are more stringent than the defaults provided by the `SSLContext` constructor, and may be adjusted in the future, without prior deprecation, if best-practice security requirements change. The new recommended best practice for using stdlib libraries that support SSL is to use `create_default_context()` to obtain an `SSLContext` object, modify it if needed, and then pass it as the *context* argument of the appropriate stdlib API. (Contributed by Christian Heimes in issue 19689.)

`SSLContext` method `load_verify_locations()` accepts a new optional argument *cadata*, which can be used to provide PEM or DER encoded certificates directly via strings or bytes, respectively. (Contributed by Christian Heimes in issue 18138.)

New function `get_default_verify_paths()` returns a named tuple of the paths and environment variables that the `set_default_verify_paths()` method uses to set OpenSSL's default `cafile` and `capath`. This can be an aid in debugging default verification issues. (Contributed by Christian Heimes in issue 18143.)

`SSLContext` has a new method, `cert_store_stats()`, that reports the number of loaded `X.509` certs, `X.509 CA` certs, and certificate revocation lists (`crl`s), as well as a `get_ca_certs()` method that returns a list of the loaded `CA` certificates. (Contributed by Christian Heimes in issue 18147.)

If OpenSSL 0.9.8 or later is available, `SSLContext` has an new attribute `verify_flags` that can be used to control the certificate verification process by setting it to some combination of the new constants `VERIFY_DEFAULT`, `VERIFY_CRL_CHECK_LEAF`, `VERIFY_CRL_CHECK_CHAIN`, or `VERIFY_X509_STRICT`. OpenSSL does not do any CRL verification by default. (Contributed by Christien Heimes in issue 8813.)

New `SSLContext` method `load_default_certs()` loads a set of default "certificate authority" (CA) certificates from default locations, which vary according to the platform. It can be used to load both TLS web server authentication certificates (`purpose=SERVER_AUTH`) for a client to use to verify a server, and certificates for a server to use in verifying client certificates (`purpose=CLIENT_AUTH`). (Contributed by Christian Heimes in issue 19292.)

Two new windows-only functions, `enum_certificates()` and `enum_crls()` provide the ability to retrieve certificates, certificate information, and CRLs from the Windows cert store. (Contributed by Christian Heimes in issue 17134.)

Support for server-side SNI (Server Name Indication) using the new `ssl.SSLContext.set_servername_callback()` method. (Contributed by Daniel Black in issue 8109.)

The dictionary returned by `SSLSocket.getpeercert()` contains additional `X509v3` extension items: `crlDistributionPoints`, `calIssuers`, and `OCSP` URIs. (Contributed by Christian Heimes in issue 18379.)

stat

The `stat` module is now backed by a C implementation in `_stat`. A C implementation is required as most of the values aren't standardized and are platform-dependent. (Contributed by Christian Heimes in issue 11016.)

The module supports new `ST_MODE` flags, `S_IFDOOR`, `S_IFPORT`, and `S_IFWHT`. (Contributed by Christian Hiemes in issue 11016.)

struct

New function `iter_unpack` and a new `struct.Struct.iter_unpack()` method on compiled formats provide streamed unpacking of a buffer containing repeated instances of a given format of data. (Contributed by Antoine Pitrou in issue 17804.)

subprocess

`check_output()` now accepts an *input* argument that can be used to provide the contents of `stdin` for the command that is run. (Contributed by Zack Weinberg in issue 16624.)

`getstatus()` and `getstatusoutput()` now work on Windows. This change was actually inadvertently made in 3.3.4. (Contributed by Tim Golden in issue 10197.)

sunau

The `getparams()` method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in issue 18901.)

`sunau.open()` now supports the context manager protocol: when used in a `with` block, the `close` method of the returned object will be called automatically at the end of the block. (Contributed by Serhiy Storchaka in issue 18878.)

`AU_write.setsampwidth()` now supports 24 bit samples, thus adding support for writing 24 sample using the module. (Contributed by Serhiy Storchaka in issue 19261.)

The `writeframesraw()` and `writeframes()` methods now accept any *bytes-like object*. (Contributed by Serhiy Storchaka in issue 8311.)

sys

New function `sys.getallocatedblocks()` returns the current number of blocks allocated by the interpreter. (In CPython with the default `--with-pymalloc` setting, this is allocations made through the `PyObject_Malloc()` API.) This can be useful for tracking memory leaks, especially if automated via a test suite. (Contributed by Antoine Pitrou in issue 13390.)

When the Python interpreter starts in *interactive mode*, it checks for an `__interactivehook__` attribute on the `sys` module. If the attribute exists, its value is called with no arguments just before interactive mode is started. The check is made after the `PYTHONSTARTUP` file is read, so it can be set there. The `site` module *sets it* to a function that enables tab completion and history saving (in `~/.python-history`) if the platform supports `readline`. If you do not want this (new) behavior, you can override it in `PYTHONSTARTUP`, `sitecustomize`, or `usercustomize` by deleting this attribute from `sys` (or setting it to some other callable). (Contributed by Éric Araujo and Antoine Pitrou in issue 5845.)

tarfile

The `tarfile` module now supports a simple *Command Line Interface* when called as a script directly or via `-m`. This can be used to create and extract tarfile archives. (Contributed by Berker Peksag in issue 13477.)

textwrap

The `TextWrapper` class has two new attributes/constructor arguments: `max_lines`, which limits the number of lines in the output, and `placeholder`, which is a string that will appear at the end of the output if it has been truncated because of *max_lines*. Building on these capabilities, a new convenience function `shorten()` collapses all of the whitespace in the input to single spaces and produces a single line of a given *width* that ends with the *placeholder* (by default, `[...]`). (Contributed by Antoine Pitrou and Serhiy Storchaka in issue 18585 and issue 18725.)

threading

The `Thread` object representing the main thread can be obtained from the new `main_thread()` function. In normal conditions this will be the thread from which the Python interpreter was started. (Contributed by Andrew Svetlov in issue 18882.)

traceback

A new `traceback.clear_frames()` function takes a traceback object and clears the local variables in all of the frames it references, reducing the amount of memory consumed. (Contributed by Andrew Kuchling in issue 1565525).

types

A new `DynamicClassAttribute()` descriptor provides a way to define an attribute that acts normally when looked up through an instance object, but which is routed to the *class* `__getattr__` when looked up through the class. This allows one to have properties active on a class, and have virtual attributes on the class with the same name (see `Enum` for an example). (Contributed by Ethan Furman in issue 19030.)

urllib

`urllib.request` now supports `data:` URLs via the `DataHandler` class. (Contributed by Mathias Panzenböck in issue 16423.)

The http method that will be used by a `Request` class can now be specified by setting a `method` class attribute on the subclass. (Contributed by Jason R Coombs in issue 18978.)

`Request` objects are now reusable: if the `full_url` or `data` attributes are modified, all relevant internal properties are updated. This means, for example, that it is now possible to use the same `Request` object in more than one `OpenerDirector.open()` call with different *data* arguments, or to modify a `Request`'s `url` rather than recomputing it from scratch. There is also a new `remove_header()` method that can be used to remove headers from a `Request`. (Contributed by Alexey Kachayev in issue 16464, Daniel Wozniak in issue 17485, and Damien Brecht and Senthil Kumaran in issue 17272.)

`HTTPError` objects now have a `headers` attribute that provides access to the HTTP response headers associated with the error. (Contributed by Berker Peksag in issue 15701.)

unittest

The TestCase class has a new method, subTest(), that produces a context manager whose with block becomes a “sub-test”. This context manager allows a test method to dynamically generate subtests by, say, calling the subTest context manager inside a loop. A single test method can thereby produce an indefinite number of separately-identified and separately-counted tests, all of which will run even if one or more of them fail. For example:

```
class NumbersTest(unittest.TestCase):
    def test_even(self):
        for i in range(6):
            with self.subTest(i=i):
                self.assertEqual(i % 2, 0)
```

will result in six subtests, each identified in the unittest verbose output with a label consisting of the variable name i and a particular value for that variable (i=0, i=1, etc). See *Distinguishing test iterations using subtests* for the full version of this example. (Contributed by Antoine Pitrou in issue 16997.)

unittest.main() now accepts an iterable of test names for *defaultTest*, where previously it only accepted a single test name as a string. (Contributed by Jyrki Pulliainen in issue 15132.)

If SkipTest is raised during test discovery (that is, at the module level in the test file), it is now reported as a skip instead of an error. (Contributed by Zach Ware in issue 16935.)

discover() now sorts the discovered files to provide consistent test ordering. (Contributed by Martin Melin and Jeff Ramnani in issue 16709.)

TestSuite now drops references to tests as soon as the test has been run, if the test is successful. On Python interpreters that do garbage collection, this allows the tests to be garbage collected if nothing else is holding a reference to the test. It is possible to override this behavior by creating a TestSuite subclass that defines a custom _removeTestAtIndex method. (Contributed by Tom Wardill, Matt McClure, and Andrew Svetlov in issue 11798.)

A new test assertion context-manager, assertLogs(), will ensure that a given block of code emits a log message using the logging module. By default the message can come from any logger and have a priority of INFO or higher, but both the logger name and an alternative minimum logging level may be specified. The object returned by the context manager can be queried for the `LogRecord`s and/or formatted messages that were logged. (Contributed by Antoine Pitrou in issue 18937.)

Test discovery now works with namespace packages (Contributed by Claudiu Popa in issue 17457.)

`unittest.mock` objects now inspect their specification signatures when matching calls, which means an argument can now be matched by either position or name, instead of only by position. (Contributed by Antoine Pitrou in issue 17015.)

`mock_open()` objects now have `readline` and `readlines` methods. (Contributed by Toshio Kuratomi in issue 17467.)

venv

`venv` now includes activation scripts for the `csh` and `fish` shells (Contributed by Andrew Svetlov in issue 15417.)

`EnvBuilder` and the `create()` convenience function take a new keyword argument *with_pip*, which defaults to `False`, that controls whether or not `EnvBuilder` ensures that `pip` is installed in the virtual environment. (Contributed by Nick Coghlan in issue 19552 as part of the **PEP 453** implementation.)

wave

The `getparams()` method now returns a namedtuple rather than a plain tuple. (Contributed by Claudiu Popa in issue 17487.)

`wave.open()` now supports the context manager protocol. (Contributed by Claudiu Popa in issue 17616.)

`wave` can now *write output to unseekable files*. (Contributed by David Jones, Guilherme Polo, and Serhiy Storchaka in issue 5202.)

The `writeframesraw()` and `writeframes()` methods now accept any *bytes-like object*. (Contributed by Serhiy Storchaka in issue 8311.)

weakref

New `WeakMethod` class simulates weak references to bound methods. (Contributed by Antoine Pitrou in issue 14631.)

New `finalize` class makes it possible to register a callback to be invoked when an object is garbage collected, without needing to carefully manage the lifecycle of the weak reference itself. (Contributed by Richard Oudkerk in issue 15528)

The callback, if any, associated with a `ref` is now exposed via the `__callback__` attribute. (Contributed by Mark Dickinson in issue 17643.)

xml.etree

A new parser, `XMLPullParser`, allows a non-blocking applications to parse XML documents. An example can be seen at *Pull API for non-blocking parsing*. (Contributed by Antoine Pitrou in issue 17741.)

The `xml.etree.ElementTree` `tostring()` and `tostringlist()` functions, and the `ElementTree` `write()` method, now have a *short_empty_elements* *keyword-only parameter* providing control over whether elements with no content are written in abbreviated (`<tag />`) or expanded (`<tag></tag>`) form. (Contributed by Ariel Poliak and Serhiy Storchaka in issue 14377.)

zipfile

The `writepy()` method of the `PyZipFile` class has a new *filterfunc* option that can be used to control which directories and files are added to the archive. For example, this could be used to exclude test files from the archive. (Contributed by Christian Tismer in issue 19274.)

The *allowZip64* parameter to `ZipFile` and `PyZipfile` is now `True` by default. (Contributed by William Mallard in issue 17201.)

CPython Implementation Changes

PEP 445: Customization of CPython Memory Allocators

PEP 445 adds new C level interfaces to customize memory allocation in the CPython interpreter.

See also:

PEP 445 – Add new APIs to customize Python memory allocators
PEP written and implemented by Victor Stinner.

PEP 442: Safe Object Finalization

PEP 442 removes the current limitations and quirks of object finalization in CPython. With it, objects with `__del__()` methods, as well as generators with `finally` clauses, can be finalized when they are part of a reference cycle.

As part of this change, module globals are no longer forcibly set to `None` during interpreter shutdown in most cases, instead relying on the normal operation of the cyclic garbage collector. This avoids a whole class of interpreter-shutdown-time errors, usually involving `__del__` methods, that have plagued Python since the cyclic GC was first introduced.

See also:

PEP 442 – Safe object finalization
PEP written and implemented by Antoine Pitrou.

PEP 456: Secure and Interchangeable Hash Algorithm

PEP 456 follows up on earlier security fix work done on Python's hash algorithm to address certain DOS attacks to which public facing APIs backed by dictionary lookups may be subject. (See issue 14621 for the start of the current round of improvements.) The PEP unifies CPython's hash code to make it easier for a packager to substitute a different hash algorithm, and switches Python's default implementation to a SipHash implementation on platforms that have a 64 bit data type. Any performance differences in comparison with the older FNV algorithm are trivial.

The PEP adds additional fields to the `sys.hash_info` struct sequence to describe the hash algorithm in use by the currently executing binary. Otherwise, the PEP does not alter any existing CPython APIs.

PEP 436: Argument Clinic

"Argument Clinic" (**PEP 436**) is now part of the CPython build process and can be used to simplify the process of defining and maintaining accurate signatures for builtins and standard library extension modules implemented in C.

Some standard library extension modules have been converted to use Argument Clinic in Python 3.4, and `pydoc` and `inspect` have been updated accordingly.

It is expected that signature metadata for programmatic introspection will be added to additional callables implemented in C as part of Python 3.4 maintenance releases.

Note: The Argument Clinic PEP is not fully up to date with the state of the implementation. This has been deemed acceptable by the release manager and core development team in this case, as Argument Clinic will not be made available as a public API for third party use in Python 3.4.

See also:

PEP 436 – The Argument Clinic DSL
PEP written and implemented by Larry Hastings.

Other Build and C API Changes

- The new `PyType_GetSlot()` function has been added to the stable ABI, allowing retrieval of function pointers from named type slots when using the limited API. (Contributed by Martin von Löwis in issue 17162)
- The new `Py_SetStandardStreamEncoding()` pre-initialization API allows applications embedding the CPython interpreter to reliably force a particular encoding and error handler for the standard streams (Contributed by Bastien Montagne and Nick Coghlan in issue 16129)
- Most Python C APIs that don't mutate string arguments are now correctly marked as accepting `const char *` rather than `char *` (Contributed by Serhiy Storchaka in issue 1772673).
- A new shell version of `python-config` can be used even when a python interpreter is not available (for example, in cross compilation scenarios).
- `PyUnicode_FromFormat()` now supports width and precision specifications for `%s`, `%A`, `%U`, `%V`, `%S`, and `%R`. (Contributed by Ysj Ray and Victor Stinner in issue 7330.)
- New function `PyStructSequence_InitType2()` supplements the existing `PyStructSequence_InitType()` function. The difference is that it returns `0` on

success and `-1` on failure.

- The CPython source can now be compiled using the address sanity checking features of recent versions of GCC and clang: the false alarms in the small object allocator have been silenced. (Contributed by Dhiru Kholia in issue 18596.)
- The Windows build now uses Address Space Layout Randomization and Data Execution Prevention. (Contributed by Christian Heimes in issue 16632.)
- New function `PyObject_LengthHint()` is the C API equivalent of `operator.length_hint()`. (Contributed by Armin Ronacher in issue 16148.)

Other Improvements

- The *python* command has a new *option*, `-I`, which causes it to run in "isolated mode", which means that `sys.path` contains neither the script's directory nor the user's `site-packages` directory, and all `PYTHON*` environment variables are ignored (it implies both `-s` and `-E`). Other restrictions may also be applied in the future, with the goal being to isolate the execution of a script from the user's environment. This is appropriate, for example, when Python is used to run a system script. On most POSIX systems it can and should be used in the `#!` line of system scripts. (Contributed by Christian Heimes in issue 16499.)
- Tab-completion is now enabled by default in the interactive interpreter on systems that support `readline`. History is also enabled by default, and is written to (and read from) the file `~/.python-history`. (Contributed by Antoine Pitrou and Éric Araujo in issue 5845.)
- Invoking the Python interpreter with `--version` now outputs the version to standard output instead of standard error (issue 18338). Similar changes were made to `argparse` (issue 18920) and other modules that have script-like invocation capabilities (issue 18922).
- The CPython Windows installer now adds `.py` to the `PATHEXT` variable when extensions are registered, allowing users to run a python script at the windows command prompt by just typing its name without the `.py` extension. (Contributed by Paul Moore in issue 18569.)
- A new `make` target coverage-report will build python, run the test suite, and generate an HTML coverage report for the C codebase using `gcov` and lcov.
- The `-R` option to the *python regression test suite* now also checks for memory allocation leaks, using `sys.getallocatedblocks()`. (Contributed by Antoine Pitrou in issue 13390).
- `python -m` now works with namespace packages.
- The `stat` module is now implemented in C, which means it gets the values for its constants from the C header files, instead of having the values hard-coded in

the python module as was previously the case.

- Loading multiple python modules from a single OS module (`.so`, `.dll`) now works correctly (previously it silently returned the first python module in the file). (Contributed by Václav Šmilauer in issue 16421.)
- A new opcode, `LOAD_CLASSDEREF`, has been added to fix a bug in the loading of free variables in class bodies that could be triggered by certain uses of *__prepare__*. (Contributed by Benjamin Peterson in issue 17853.)
- A number of MemoryError-related crashes were identified and fixed by Victor Stinner using his **PEP 445**-based `pyfailmalloc` tool (issue 18408, issue 18520).
- The *pyvenv* command now accepts a `--copies` option to use copies rather than symlinks even on systems where symlinks are the default. (Contributed by Vinay Sajip in issue 18807.)
- The *pyvenv* command also accepts a `--without-pip` option to suppress the otherwise-automatic bootstrapping of pip into the virtual environment. (Contributed by Nick Coghlan in issue 19552 as part of the **PEP 453** implementation.)
- The encoding name is now optional in the value set for the `PYTHONIOENCODING` environment variable. This makes it possible to set just the error handler, without changing the default encoding. (Contributed by Serhiy Storchaka in issue 18818.)
- The `bz2`, `lzma`, and `gzip` module `open` functions now support `x` (exclusive creation) mode. (Contributed by Tim Heaney and Vajrasky Kok in issue 19201, issue 19222, and issue 19223.)

Significant Optimizations

- The UTF-32 decoder is now 3x to 4x faster. (Contributed by Serhiy Storchaka in issue 14625.)
- The cost of hash collisions for sets is now reduced. Each hash table probe now checks a series of consecutive, adjacent key/hash pairs before continuing to make random probes through the hash table. This exploits cache locality to make collision resolution less expensive. The collision resolution scheme can be described as a hybrid of linear probing and open addressing. The number of additional linear probes defaults to nine. This can be changed at compile-time by defining LINEAR_PROBES to be any value. Set LINEAR_PROBES=0 to turn-off linear probing entirely. (Contributed by Raymond Hettinger in issue 18771.)
- The interpreter starts about 30% faster. A couple of measures lead to the speedup. The interpreter loads fewer modules on startup, e.g. the `re`, `collections` and `locale` modules and their dependencies are no longer imported by default. The marshal module has been improved to load compiled Python code faster. (Contributed by Antoine Pitrou, Christian Heimes and Victor

Stinner in issue 19219, issue 19218, issue 19209, issue 19205 and issue 9548)

- `bz2.BZ2File` is now as fast or faster than the Python2 version for most cases. `lzma.LZMAFile` has also been optimized. (Contributed by Serhiy Storchaka and Nadeem Vawda in issue 16034.)
- `random.getrandbits()` is 20%-40% faster for small integers (the most common use case). (Contributed by Serhiy Storchaka in issue 16674).
- By taking advantage of the new storage format for strings, pickling of strings is now significantly faster. (Contributed by Victor Stinner and Antoine Pitrou in issue 15596.)
- A performance issue in `io.FileIO.readall()` has been solved. This particularly affects Windows, and significantly speeds up the case of piping significant amounts of data through `subprocess`. (Contributed by Richard Oudkerk in issue 15758.)
- `html.escape()` is now 10x faster. (Contributed by Matt Bryant in issue 18020.)
- On Windows, the native `VirtualAlloc` is now used instead of the CRT `malloc` in `obmalloc`. Artificial benchmarks show about a 3% memory savings.
- `os.urandom()` now uses a lazily-opened persistent file descriptor so as to avoid using many file descriptors when run in parallel from multiple threads. (Contributed by Antoine Pitrou in issue 18756.)

Deprecated

This section covers various APIs and other features that have been deprecated in Python 3.4, and will be removed in Python 3.5 or later. In most (but not all) cases, using the deprecated APIs will produce a `DeprecationWarning` when the interpreter is run with deprecation warnings enabled (for example, by using `-Wd`).

Deprecations in the Python API

- As mentioned in *PEP 451: A ModuleSpec Type for the Import System*, a number of `importilb` methods and functions are deprecated: `importlib.find_loader()` is replaced by `importlib.util.find_spec()`; `importlib.machinery.PathFinder.find_module()` is replaced by `importlib.machinery.PathFinder.find_spec()`; `importlib.abc.MetaPathFinder.find_module()` is replaced by `importlib.abc.MetaPathFinder.find_spec()`; `importlib.abc.PathEntryFinder.find_loader()` and `find_module()` are replaced by `importlib.abc.PathEntryFinder.find_spec()`; all of the `xxxLoader` ABC `load_module` methods (`importlib.abc.Loader.load_module()`, importlib.abc.InspectLoader.load_module(), importlib.abc.FileLoader.load_module(), importlib.abc.SourceLoader.load_module()) should no longer be implemented, instead loaders should implement an `exec_module` method (importlib.abc.Loader.exec_module(), importlib.abc.InspectLoader.exec_module() importlib.abc.SourceLoader.exec_module()) and let the import system take care of the rest; and importlib.abc.Loader.module_repr(), importlib.util.module_for_loader(), importlib.util.set_loader(), and importlib.util.set_package() are no longer needed because their functions are now handled automatically by the import system.

- The imp module is pending deprecation. To keep compatibility with Python 2/3 code bases, the module's removal is currently not scheduled.
- The formatter module is pending deprecation and is slated for removal in Python 3.6.
- `MD5` as the default *digestmod* for the hmac.new() function is deprecated. Python 3.6 will require an explicit digest name or constructor as *digestmod* argument.
- The internal `Netrc` class in the ftplib module has been documented as deprecated in its docstring for quite some time. It now emits a DeprecationWarning and will be removed completely in Python 3.5.
- The undocumented *endtime* argument to subprocess.Popen.wait() should not have been exposed and is hopefully not in use; it is deprecated and will mostly likely be removed in Python 3.5.
- The *strict* argument of HTMLParser is deprecated.
- The plistlib readPlist(), writePlist(), readPlistFromBytes(), and writePlistToBytes() functions are deprecated in favor of the corresponding new functions load(), dump(), loads(), and dumps(). Data() is deprecated in favor of just using the bytes constructor.
- The sysconfig key `SO` is deprecated, it has been replaced by `EXT_SUFFIX`.
- The `U` mode accepted by various `open` functions is deprecated. In Python3 it does not do anything useful, and should be replaced by appropriate uses of io.TextIOWrapper (if needed) and its *newline* argument.
- The *parser* argument of xml.etree.ElementTree.iterparse() has been deprecated, as has the *html* argument of XMLParser(). To prepare for the removal of the latter, all arguments to `XMLParser` should be passed by keyword.

Deprecated Features

- Running *IDLE* with the `-n` flag (no subprocess) is deprecated. However, the

feature will not be removed until issue 18823 is resolved.
- The site module adding a "site-python" directory to sys.path, if it exists, is deprecated (issue 19375).

Removed

Operating Systems No Longer Supported

Support for the following operating systems has been removed from the source and build tools:

- OS/2 (issue 16135).
- Windows 2000 (changeset e52df05b496a).
- Windows systems where `COMSPEC` points to `command.com` (issue 14470).
- VMS (issue 16136).

API and Feature Removals

The following obsolete and previously deprecated APIs and features have been removed:

- The unmaintained `Misc/TextMate` and `Misc/vim` directories have been removed (see the devguide for suggestions on what to use instead).
- The `SO` makefile macro is removed (it was replaced by the `SHLIB_SUFFIX` and `EXT_SUFFIX` macros) (issue 16754).
- The `PyThreadState.tick_counter` field has been removed; its value has been meaningless since Python 3.2, when the "new GIL" was introduced (issue 19199).
- `PyLoader` and `PyPycLoader` have been removed from `importlib`. (Contributed by Taras Lyapun in issue 15641.)
- The *strict* argument to `HTTPConnection` and `HTTPSConnection` has been removed. HTTP 0.9-style "Simple Responses" are no longer supported.
- The deprecated `urllib.request.Request` getter and setter methods `add_data`, `has_data`, `get_data`, `get_type`, `get_host`, `get_selector`, `set_proxy`, `get_origin_req_host`, and `is_unverifiable` have been removed (use direct attribute access instead).
- Support for loading the deprecated `TYPE_INT64` has been removed from `marshal`. (Contributed by Dan Riti in issue 15480.)
- `inspect.Signature`: positional-only parameters are now required to have a valid name.

- `object.__format__()` no longer accepts non-empty format strings, it now raises a `TypeError` instead. Using a non-empty string has been deprecated since Python 3.2. This change has been made to prevent a situation where previously working (but incorrect) code would start failing if an object gained a __format__ method, which means that your code may now raise a `TypeError` if you are using an `'s'` format code with objects that do not have a __format__ method that handles it. See issue 7994 for background.
- `difflib.SequenceMatcher.isbjunk()` and `difflib.SequenceMatcher.isbpopular()` were deprecated in 3.2, and have now been removed: use `x in sm.bjunk` and `x in sm.bpopular`, where *sm* is a `SequenceMatcher` object (issue 13248).

Code Cleanups

- The unused and undocumented internal `Scanner` class has been removed from the `pydoc` module.
- The private and effectively unused `_gestalt` module has been removed, along with the private `platform` functions `_mac_ver_lookup`, `_mac_ver_gstalt`, and `_bcd2str`, which would only have ever been called on badly broken OSX systems (see issue 18393).
- The hardcoded copies of certain `stat` constants that were included in the `tarfile` module namespace have been removed.

Porting to Python 3.4

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

Changes in 'python' Command Behavior

- In a posix shell, setting the `PATH` environment variable to an empty value is equivalent to not setting it at all. However, setting `PYTHONPATH` to an empty value was *not* equivalent to not setting it at all: setting `PYTHONPATH` to an empty value was equivalent to setting it to `.`, which leads to confusion when reasoning by analogy to how `PATH` works. The behavior now conforms to the posix convention for `PATH`.
- The [X refs, Y blocks] output of a debug (`--with-pydebug`) build of the CPython interpreter is now off by default. It can be re-enabled using the `-X`

`showrefcount` option. (Contributed by Ezio Melotti in issue 17323.)
- The python command and most stdlib scripts (as well as `argparse`) now output `--version` information to `stdout` instead of `stderr` (for issue list see *Other Improvements* above).

Changes in the Python API

- The ABCs defined in `importlib.abc` now either raise the appropriate exception or return a default value instead of raising `NotImplementedError` blindly. This will only affect code calling `super()` and falling through all the way to the ABCs. For compatibility, catch both `NotImplementedError` or the appropriate exception as needed.
- The module type now initializes the `__package__` and `__loader__` attributes to `None` by default. To determine if these attributes were set in a backwards-compatible fashion, use e.g. `getattr(module, '__loader__', None) is not None`. (issue 17115.)
- `importlib.util.module_for_loader()` now sets `__loader__` and `__package__` unconditionally to properly support reloading. If this is not desired then you will need to set these attributes manually. You can use `importlib.util.module_to_load()` for module management.
- Import now resets relevant attributes (e.g. `__name__`, `__loader__`, `__package__`, `__file__`, `__cached__`) unconditionally when reloading. Note that this restores a pre-3.3 behavior in that it means a module is re-found when re-loaded (issue 19413).
- Frozen packages no longer set `__path__` to a list containing the package name, they now set it to an empty list. The previous behavior could cause the import system to do the wrong thing on submodule imports if there was also a directory with the same name as the frozen package. The correct way to determine if a module is a package or not is to use `hasattr(module, '__path__')` (issue 18065).
- Frozen modules no longer define a `__file__` attribute. It's semantically incorrect for frozen modules to set the attribute as they are not loaded from any explicit location. If you must know that a module comes from frozen code then you can see if the module's `__spec__.location` is set to `'frozen'`, check if the loader is a subclass of `importlib.machinery.FrozenImporter`, or if Python 2 compatibility is necessary you can use `imp.is_frozen()`.
- `py_compile.compile()` now raises `FileExistsError` if the file path it would write to is a symlink or a non-regular file. This is to act as a warning that import will overwrite those files with a regular file regardless of what type of file path they were originally.

- `importlib.abc.SourceLoader.get_source()` no longer raises `ImportError` when the source code being loaded triggers a `SyntaxError` or `UnicodeDecodeError`. As `ImportError` is meant to be raised only when source code cannot be found but it should, it was felt to be over-reaching/overloading of that meaning when the source code is found but improperly structured. If you were catching ImportError before and wish to continue to ignore syntax or decoding issues, catch all three exceptions now.
- `functools.update_wrapper()` and `functools.wraps()` now correctly set the `__wrapped__` attribute to the function being wrapped, even if that function also had its `__wrapped__` attribute set. This means `__wrapped__` attributes now correctly link a stack of decorated functions rather than every `__wrapped__` attribute in the chain referring to the innermost function. Introspection libraries that assumed the previous behaviour was intentional can use `inspect.unwrap()` to access the first function in the chain that has no `__wrapped__` attribute.
- `inspect.getfullargspec()` has been reimplemented on top of `inspect.signature()` and hence handles a much wider variety of callable objects than it did in the past. It is expected that additional builtin and extension module callables will gain signature metadata over the course of the Python 3.4 series. Code that assumes that `inspect.getfullargspec()` will fail on non-Python callables may need to be adjusted accordingly.
- `importlib.machinery.PathFinder` now passes on the current working directory to objects in `sys.path_hooks` for the empty string. This results in `sys.path_importer_cache` never containing `''`, thus iterating through `sys.path_importer_cache` based on `sys.path` will not find all keys. A module's `__file__` when imported in the current working directory will also now have an absolute path, including when using `-m` with the interpreter (except for `__main__.__file__` when a script has been executed directly using a relative path) (Contributed by Brett Cannon in issue 18416). is specified on the command-line) (issue 18416).
- The removal of the *strict* argument to `HTTPConnection` and `HTTPSConnection` changes the meaning of the remaining arguments if you are specifying them positionally rather than by keyword. If you've been paying attention to deprecation warnings your code should already be specifying any additional arguments via keywords.
- Strings between `from __future__ import ...` statements now *always* raise a `SyntaxError`. Previously if there was no leading docstring, an interstitial string would sometimes be ignored. This brings CPython into compliance with the language spec; Jython and PyPy already were. (issue 17434).
- `ssl.SSLSocket.getpeercert()` and `ssl.SSLSocket.do_handshake()` now raise an `OSError` with `ENOTCONN` when the `SSLSocket` is not connected, instead of the

previous behavior of raising an `AttributeError`. In addition, `getpeercert()` will raise a `ValueError` if the handshake has not yet been done.

- `base64.b32decode()` now raises a `binascii.Error` when the input string contains non-b32-alphabet characters, instead of a `TypeError`. This particular `TypeError` was missed when the other `TypeError`s were converted. (Contributed by Serhiy Storchaka in issue 18011.) Note: this change was also inadvertently applied in Python 3.3.3.
- The `file` attribute is now automatically closed when the creating `cgi.FieldStorage` instance is garbage collected. If you were pulling the file object out separately from the `cgi.FieldStorage` instance and not keeping the instance alive, then you should either store the entire `cgi.FieldStorage` instance or read the contents of the file before the `cgi.FieldStorage` instance is garbage collected.
- Calling `read` or `write` on a closed SSL socket now raises an informative `ValueError` rather than the previous more mysterious `AttributeError` (issue 9177).
- `slice.indices()` no longer produces an `OverflowError` for huge values. As a consequence of this fix, `slice.indices()` now raises a `ValueError` if given a negative length; previously it returned nonsense values (issue 14794).
- The `complex` constructor, unlike the `cmath` functions, was incorrectly accepting `float` values if an object's `__complex__` special method returned one. This now raises a `TypeError`. (issue 16290.)
- The `int` constructor in 3.2 and 3.3 erroneously accepts `float` values for the *base* parameter. It is unlikely anyone was doing this, but if so, it will now raise a `TypeError` (issue 16772).
- Defaults for keyword-only arguments are now evaluated *after* defaults for regular keyword arguments, instead of before. Hopefully no one wrote any code that depends on the previous buggy behavior (issue 16967).
- Stale thread states are now cleared after `fork()`. This may cause some system resources to be released that previously were incorrectly kept perpetually alive (for example, database connections kept in thread-local storage). (issue 17094.)
- Parameter names in `__annotations__` dicts are now mangled properly, similarly to `__kwdefaults__`. (Contributed by Yury Selivanov in issue 20625).
- `hashlib.hash.name` now always returns the identifier in lower case. Previously some builtin hashes had uppercase names, but now that it is a formal public interface the naming has been made consistent (issue 18532).
- Because `unittest.TestSuite` now drops references to tests after they are run, test harnesses that re-use a `TestSuite` to re-run a set of tests may fail. Test suites should not be re-used in this fashion since it means state is retained between test runs, breaking the test isolation that `unittest` is designed to

provide. However, if the lack of isolation is considered acceptable, the old behavior can be restored by creating a `TestSuite` subclass that defines a `_removeTestAtIndex` method that does nothing (see `TestSuite.__iter__()`) (issue 11798).

- `unittest` now uses `argparse` for command line parsing. There are certain invalid command forms that used to work that are no longer allowed; in theory this should not cause backward compatibility issues since the disallowed command forms didn't make any sense and are unlikely to be in use.
- The `re.split()`, `re.findall()`, and `re.sub()` functions, and the `group()` and `groups()` methods of `match` objects now always return a *bytes* object when the string to be matched is a *bytes-like object*. Previously the return type matched the input type, so if your code was depending on the return value being, say, a `bytearray`, you will need to change your code.
- `audioop` functions now raise an error immediately if passed string input, instead of failing randomly later on (issue 16685).
- The new *convert_charrefs* argument to `HTMLParser` currently defaults to `False` for backward compatibility, but will eventually be changed to default to `True`. It is recommended that you add this keyword, with the appropriate value, to any `HTMLParser` calls in your code (issue 13633).
- Since the *digestmod* argument to the `hmac.new()` function will in the future have no default, all calls to `hmac.new()` should be changed to explicitly specify a *digestmod* (issue 17276).
- Calling `sysconfig.get_config_var()` with the `SO` key, or looking `SO` up in the results of a call to `sysconfig.get_config_vars()` is deprecated. This key should be replaced by `EXT_SUFFIX` or `SHLIB_SUFFIX`, depending on the context (issue 19555).
- Any calls to `open` functions that specify `U` should be modified. `U` is ineffective in Python3 and will eventually raise an error if used. Depending on the function, the equivalent of its old Python2 behavior can be achieved using either a *newline* argument, or if necessary by wrapping the stream in `TextIOWrapper` to use its *newline* argument (issue 15204).
- If you use *pyvenv* in a script and desire that pip *not* be installed, you must add `--without-pip` to your command invocation.
- The default behavior of `json.dump()` and `json.dumps()` when an indent is specified has changed: it no longer produces trailing spaces after the item separating commas at the ends of lines. This will matter only if you have tests that are doing white-space-sensitive comparisons of such output (issue 16333).
- `doctest` now looks for doctests in extension module `__doc__` strings, so if your doctest test discovery includes extension modules that have things that look like doctests in them you may see test failures you've never seen before when

running your tests (issue 3158).
- The `collections.abc` module has been slightly refactored as part of the Python startup improvements. As a consequence of this, it is no longer the case that importing `collections` automatically imports `collections.abc`. If your program depended on the (undocumented) implicit import, you will need to add an explicit `import collections.abc` (issue 20784).

Changes in the C API

- `PyEval_EvalFrameEx()`, `PyObject_Repr()`, and `PyObject_Str()`, along with some other internal C APIs, now include a debugging assertion that ensures they are not used in situations where they may silently discard a currently active exception. In cases where discarding the active exception is expected and desired (for example, because it has already been saved locally with `PyErr_Fetch()` or is being deliberately replaced with a different exception), an explicit `PyErr_Clear()` call will be needed to avoid triggering the assertion when invoking these operations (directly or indirectly) and running against a version of Python that is compiled with assertions enabled.
- `PyErr_SetImportError()` now sets `TypeError` when its **msg** argument is not set. Previously only `NULL` was returned with no exception set.
- The result of the `PyOS_ReadlineFunctionPointer` callback must now be a string allocated by `PyMem_RawMalloc()` or `PyMem_RawRealloc()`, or *NULL* if an error occurred, instead of a string allocated by `PyMem_Malloc()` or `PyMem_Realloc()` (issue 16742)
- `PyThread_set_key_value()` now always set the value. In Python 3.3, the function did nothing if the key already exists (if the current value is a non-NULL pointer).
- The `f_tstate` (thread state) field of the `PyFrameObject` structure has been removed to fix a bug: see issue 14432 for the rationale.